Generic Generation of Constrained Random Data

José Pedro Magalhães

Department of Computer Science
University of Oxford

March 25, 2014

Random data generation: why?

Being able to generate random data is useful:

- Testing

Random data generation: why?

Being able to generate random data is useful:

- Testing
- Examples

Random data generation: why?

Being able to generate random data is useful:

- Testing
- Examples
- In complex domains:

Random data generation: why?

Being able to generate random data is useful:

- Testing
- Examples
- In complex domains:
  - Exercises in an exercise assistant

Random data generation: why?

Being able to generate random data is useful:

- Testing
- Examples
- In complex domains:
  - Exercises in an exercise assistant
  - Population for genetic/evolutionary algorithms

Random data generation: why?

Being able to generate random data is useful:

- Testing
- Examples
- In complex domains:
  - Exercises in an exercise assistant
  - Population for genetic/evolutionary algorithms
  - Music sequences in a music modelling tool

Random data generation: why?

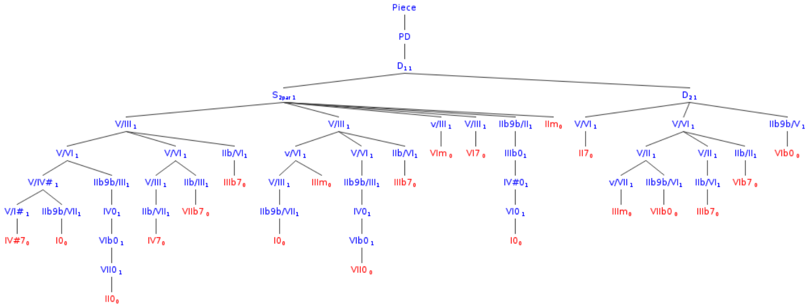
Being able to generate random data is useful:

- Testing
- Examples
- In complex domains:
  - Exercises in an exercise assistant
  - Population for genetic/evolutionary algorithms
  - Music sequences in a music modelling tool

Random data generation: why?

Being able to generate random data is useful:

- Testing
- Examples
- In complex domains:
  - Exercises in an exercise assistant
  - Population for genetic/evolutionary algorithms
  - Music sequences in a music modelling tool



Generic random data generation: why?

Being able to generate random data *generically* is useful too:

- Data generators can be tedious to write; if they're generic, we don't have to write them.

Generic random data generation: why?

Being able to generate random data *generically* is useful too:

- Data generators can be tedious to write; if they're generic, we don't have to write them.
- Datatypes change, and the code to generate them must change too—unless it's generic.

Generic random data generation: why?

Being able to generate random data *generically* is useful too:

- Data generators can be tedious to write; if they're generic, we don't have to write them.
- Datatypes change, and the code to generate them must change too—unless it's generic.
- Good generators can be hard to get right; they have to take into account the size of the generated terms, ensure a fair distribution, etc. If we do it generically, the hard work is done "once and for all".

Generic, but constrained, random data generation

Generic programs can be written because they are agnostic about the semantics of the data, and work in the same way for the same structure.

Generic, but constrained, random data generation

Generic programs can be written because they are agnostic about the semantics of the data, and work in the same way for the same structure.

Unfortunately, data generation is very often tied to semantics, especially for complex data:

- Generating sorted lists
- Generating balanced trees
- Generating good exercises (not too easy, not too hard)
- Generating valid programs

Generic, but constrained, random data generation

Generic programs can be written because they are agnostic about the semantics of the data, and work in the same way for the same structure.

Unfortunately, data generation is very often tied to semantics, especially for complex data:

- Generating sorted lists
- Generating balanced trees
- Generating good exercises (not too easy, not too hard)
- Generating valid programs

What we need is a hybrid solution, where generic programming does the "boring parts", but still allows us to specify constraints on the shape of the data generated.

Easy way: generate random data, then filter

Assuming we can just generically generate random data, we could then simply *filter* the data according to some predicate.

Easy way: generate random data, then filter

Assuming we can just generically generate random data, we could then simply *filter* the data according to some predicate.

In practice, however, this can be terribly inefficient, especially if the interesting data is a very small subset of all possible data. So we don't want to take this approach (but see Claessen et al.,"Generating Constrained Random Data with Uniform Distribution" @ FLOPS'14).

Outline

We're aiming at something relatively complicated, so we'll go step-by-step:

1. Generating random data generically, no constraints

Outline

We're aiming at something relatively complicated, so we'll go step-by-step:

1. Generating random data generically, no constraints
2. Constraining constructor frequency

Outline

We're aiming at something relatively complicated, so we'll go step-by-step:

1. Generating random data generically, no constraints
2. Constraining constructor frequency
3. Constraining size and order

Outline

We're aiming at something relatively complicated, so we'll go step-by-step:

1. Generating random data generically, no constraints
2. Constraining constructor frequency
3. Constraining size and order
4. Constraining constructor count

Outline

We're aiming at something relatively complicated, so we'll go step-by-step:

1. Generating random data generically, no constraints
2. Constraining constructor frequency
3. Constraining size and order
4. Constraining constructor count

Outline

We're aiming at something relatively complicated, so we'll go step-by-step:

1. Generating random data generically, no constraints
2. Constraining constructor frequency
3. Constraining size and order
4. Constraining constructor count

What we will not see:

- Efficiency concerns
- Nested datatypes, GADTs, etc.

Generic programming support: universe

We need a library for generic programming:

$$
\begin{array}{l}
\textbf{kind}\ \mathit{Univ} = \\
\quad\ \ \mathit{Data\ Symbol\ Univ} \\
\quad \mid \mathit{Con\ Symbol\ Univ} \\
\quad \mid \mathit{Unit} \\
\quad \mid \mathit{K\ ArgType} \star \\
\quad \mid \mathit{Univ} :\times: \mathit{Univ} \\
\quad \mid \mathit{Univ} :+: \mathit{Univ}
\end{array}
$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ [\![\, \upsilon :: \mathit{Univ} \,]\!] :: \star\ \textbf{where}} \\
\quad \mathit{D} & :: [\![\, \upsilon \,]\!] & & \rightarrow [\![\, \mathit{Data}\ \iota\ \upsilon \,]\!] \\
\quad \mathit{C} & :: [\![\, \upsilon \,]\!] & & \rightarrow [\![\, \mathit{Con}\ \iota\ \upsilon \,]\!] \\
\quad \mathit{Unit} & :: & & [\![\, \mathit{Unit} \,]\!] \\
\quad \mathit{K} & :: \alpha & & \rightarrow [\![\, \mathit{K}\ \iota\ \alpha \,]\!] \\
\quad (:\times:) & :: [\![\, \phi \,]\!] & \rightarrow [\![\, \psi \,]\!] & \rightarrow [\![\, \phi :\times: \psi \,]\!] \\
\quad \mathit{Left} & :: [\![\, \phi \,]\!] & & \rightarrow [\![\, \phi :+: \psi \,]\!] \\
\quad \mathit{Right} & :: [\![\, \psi \,]\!] & & \rightarrow [\![\, \phi :+: \psi \,]\!]
\end{array}
$$

$$
\begin{array}{l}
\textbf{kind}\ \mathit{ArgType} = \mathit{Par} \mid \mathit{Rec\ RecType} \mid \mathit{Unknown} \\
\textbf{kind}\ \mathit{RecType} = \mathit{Self} \mid \mathit{Other}
\end{array}
$$

Generic programming support: *to*/*from*

Witnessing the isomorphism between a datatype and its generic representation:

```
class Generic (α :: ⋆) where
  type Rep α :: Univ
  from :: α → ⟦ Rep α ⟧
  to   :: ⟦ Rep α ⟧ → α
```

Generic programming support: *to*/*from*

Witnessing the isomorphism between a datatype and its generic representation:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Generic}\ (\alpha :: \star)\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Rep}\ \alpha :: \mathit{Univ} \\
\quad \mathit{from} :: \alpha \to [\![\, \mathit{Rep}\ \alpha \,]\!] \\
\quad \mathit{to}\ \ \ \ :: [\![\, \mathit{Rep}\ \alpha \,]\!] \to \alpha
\end{array}
$$

For example, lists:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Generic}\ [\,\alpha\,]\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Rep}\ [\,\alpha\,] = \mathit{Data}\ \texttt{"[]"}\ (\quad \mathit{Con}\ \texttt{"[]"}\ \mathit{Unit} \\
\qquad\qquad\qquad\qquad\qquad\quad :\!+\!:\ \mathit{Con}\ \texttt{":"}\ \ (\quad K\ \mathit{Par}\qquad\quad \alpha \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad :\!\times\!:\ K\ (\mathit{Rec}\ \mathit{Self})\ [\,\alpha\,])) \\
\quad \mathit{from}\ [\,]\qquad\ = D\ (\mathit{Left}\ (C\ \mathit{Unit})) \\
\quad \mathit{from}\ (h : t) = D\ (\mathit{Right}\ (C\ (K\ h :\!\times\!: K\ t))) \\
\quad \mathit{to}\ (D\ (\mathit{Left}\quad (C\ \mathit{Unit})))\qquad\qquad = [\,] \\
\quad \mathit{to}\ (D\ (\mathit{Right}\ (C\ (K\ h :\!\times\!: K\ t)))) = h : t
\end{array}
$$

Section 1

Generic data generation, no constraints

Generic data generation, no constraints: I

We need a user-facing class (just like *Arbitrary*):

```
class Generate α where
  gen :: Gen α
```

Generic data generation, no constraints: I

We need a user-facing class (just like *Arbitrary*):

```
class Generate α where
  gen :: Gen α
```

And one to define data generation generically:

```
class GGenerate (υ :: Univ) where
  gGen :: Gen (⟦ υ ⟧)
```

Generic data generation, no constraints: II

Now we can define the generic instances:

```
instance (GGenerate υ) ⇒ GGenerate (Data ι υ) where
  gGen = fmap D gGen
instance (GGenerate φ, GGenerate ψ) ⇒ GGenerate (φ :+: ψ) where
  gGen = choose [fmap Left gGen, fmap Right gGen]
instance (GGenerate υ) ⇒ GGenerate (Con ι υ) where
  gGen = fmap C gGen
instance GGenerate Unit where
  gGen = return Unit
instance (GGenerate φ, GGenerate ψ) ⇒ GGenerate (φ :×: ψ) where
  gGen = (:×:) <$> gGen <∗> gGen
instance (Generate α) ⇒ GGenerate (K ι α) where
  gGen = fmap K gen
```

Generic data generation, no constraints: III

Finally, a generic dispatcher:

$$\begin{array}{l} \mathit{genDefault} :: (\mathit{Generic}\ \alpha, \mathit{GGenerate}\ (\mathit{Rep}\ \alpha)) \Rightarrow \mathit{Gen}\ \alpha \\ \mathit{genDefault} = \mathit{fmap}\ \mathit{to}\ \mathit{gGen} \end{array}$$

Generic data generation, no constraints: III

Finally, a generic dispatcher:

$$\begin{aligned}&\mathit{genDefault} :: (\mathit{Generic}\ \alpha, \mathit{GGenerate}\ (\mathit{Rep}\ \alpha)) \Rightarrow \mathit{Gen}\ \alpha\\ &\mathit{genDefault} = \mathit{fmap}\ \mathit{to}\ \mathit{gGen}\end{aligned}$$

And we're ready to generically generate random, unconstrained data.

```
> sample (genDefault :: [Bool])
[]
[True, False, True]
[]
[False]
[]
[True, True]
[]
[False]
```

Section 2

Constructor frequency constraints

Constructor frequency constraints: I

Now let's start considering constraints. First, we want to be able to say that some constructors should occur more/less often than others.

We will need to keep track of some information during generation:

```
data MyState = MyState { envState    :: Env        -- environment
                       , dtNameState :: String }   -- datatype name
type Env     = Map ConLoc ConInfo
type ConLoc  = (String    -- datatype name
               , String)  -- constructor name
data ConInfo = ConInfo { freqInfo :: Int }
```

Constructor frequency constraints: II

We adapt the type of our functions:

$\textbf{class}\ \mathit{Generate}\ \alpha\ \textbf{where}$
$\quad \mathit{gen} :: \mathit{MyState} \rightarrow \mathit{Gen}\ \alpha$

$\textbf{class}\ \mathit{GGenerate}\ (\upsilon :: \mathit{Univ})\ \textbf{where}$
$\quad \mathit{gGen} :: \mathit{MyState} \rightarrow \mathit{Gen}\ (\llbracket\, \upsilon\, \rrbracket)$

Constructor frequency constraints: II

We adapt the type of our functions:

```
class Generate α where
  gen   :: MyState → Gen α
class GGenerate (υ :: Univ) where
  gGen :: MyState → Gen (⟦ υ ⟧)
```

And each of the generic cases:

```
instance (SingI ι, GGenerate υ) ⇒ GGenerate (Data ι υ) where
  gGen s = fmap D (gGen (s { dtNameState = fromSing (sing :: Sing ι) }))
instance (GGenerate υ) ⇒ GGenerate (Con ι υ) where
  gGen = fmap C ∘ gGen
instance GGenerate Unit where
  gGen _ = return Unit
instance (GGenerate φ, GGenerate ψ) ⇒ GGenerate (φ :×: ψ) where
  gGen s = (:×:) <$> gGen s <*> gGen s
instance (Generate α) ⇒ GGenerate (K ι α) where
  gGen = fmap K ∘ gen
```

Constructor frequency constraints: III

The interesting case is that of sums:

```
instance (GGenerate α, GConNames α
         , GGenerate β, GConNames β) ⇒ GGenerate (α :+: β) where
  gGen s = let aNames = gconNames (Proxy :: Proxy (⟦ α ⟧))
               bNames = gconNames (Proxy :: Proxy (⟦ β ⟧))
               dtName = dtNameState s
               env    = envState s
               aFreq  = frequencies aNames dtName env
               bFreq  = frequencies bNames dtName env
           in frequency [( aFreq, fmap Left  (gGen s))
                        ,( bFreq, fmap Right (gGen s))]

frequencies :: [String] → String → Env → Int
frequencies []       _      _   = 0
frequencies (s : ss) dtName env =
  let freqS = maybe 1 freqInfo (lookup (dtName, s) env)
  in freqS + frequencies ss dtName env

frequency :: [(Int, Gen α)] → Gen α
```

Constructor frequency constraints: IV

We need to be able to collect the names of all constructors of a datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Proxy}\ \alpha = \mathit{Proxy} \\
\textbf{class}\ \mathit{GConNames}\ (\upsilon :: \mathit{Univ})\ \textbf{where} \\
\quad \mathit{gconNames} :: \mathit{Proxy}\ (\llbracket\ \upsilon\ \rrbracket) \to [\mathit{String}] \\
\textbf{instance}\ (\mathit{GConNames}\ \upsilon) \Rightarrow \mathit{GConNames}\ (\mathit{Data}\ \iota\ \upsilon)\ \textbf{where} \\
\quad \mathit{gconNames}\ \_ = \mathit{gconNames}\ (\mathit{Proxy} :: \mathit{Proxy}\ (\llbracket\ \upsilon\ \rrbracket)) \\
\textbf{instance}\ (\mathit{GConNames}\ \phi, \mathit{GConNames}\ \psi) \Rightarrow \mathit{GConNames}\ (\phi :+: \psi)\ \textbf{where} \\
\quad \mathit{gconNames}\ \_ = \quad\ \mathit{gconNames}\ (\mathit{Proxy} :: \mathit{Proxy}\ (\llbracket\ \phi\ \rrbracket)) \\
\qquad\qquad\qquad\quad +\!\!+ \mathit{gconNames}\ (\mathit{Proxy} :: \mathit{Proxy}\ (\llbracket\ \psi\ \rrbracket)) \\
\textbf{instance}\ (\mathit{SingI}\ \iota) \Rightarrow \mathit{GConNames}\ (\mathit{Con}\ \iota\ \upsilon)\ \textbf{where} \\
\quad \mathit{gconNames}\ \_ = [\mathit{fromSing}\ (\mathit{sing} :: \mathit{Sing}\ \iota)] \\
\\
\textbf{class}\ \mathit{ConNames}\ (\alpha :: \star)\ \textbf{where} \\
\quad \mathit{conNames} :: \mathit{Proxy}\ \alpha \to [\mathit{String}]
\end{array}
$$

Constructor frequency constraints: V

Now we can constrain constructor frequency!

```
data Choice = A | B | C | D

testChoice :: IO ()
testChoice = let s      = MyState (fromList reqEnv) ""
                 reqEnv = [(("Choice","A"), ConInfo 1)
                          ,(("Choice","B"), ConInfo 2)
                          ,(("Choice","C"), ConInfo 3)
                          ,(("Choice","D"), ConInfo 4)]
             in testM (genDefault s :: Gen Choice) 1000 >>= mapM_ print

testM :: Gen α → Int → IO [(Int, α)]
  -- takes a generator, generates n samples, groups them
```

Constructor frequency constraints: V

Now we can constrain constructor frequency!

```
data Choice = A | B | C | D

testChoice :: IO ()
testChoice = let s       = MyState (fromList reqEnv) ""
                 reqEnv = [(("Choice","A"), ConInfo 1)
                          ,(("Choice","B"), ConInfo 2)
                          ,(("Choice","C"), ConInfo 3)
                          ,(("Choice","D"), ConInfo 4)]
             in testM (genDefault s :: Gen Choice) 1000 >>= mapM_ print

testM :: Gen α → Int → IO [(Int, α)]
  -- takes a generator, generates n samples, groups them

> testChoice
(99,  A)
(212, B)
(314, C)
(376, D)
```

Constructor frequency constraints: VI

We can even forbid certain constructors, and (indirectly) influence size:

```
testListBool :: IO ()
testListBool = let s      = MyState (fromList reqEnv) ""
                   reqEnv = [(("Bool","True")  , ConInfo 1)
                            ,(("Bool","False"), ConInfo 0)
                            ,(("[]",":")       , ConInfo 3)
                            ,(("[]","[]")      , ConInfo 1)]
               in testM (genDefault s :: Gen [Bool]) 10 >>= mapM_ print
```

Constructor frequency constraints: VI

We can even forbid certain constructors, and (indirectly) influence size:

```
testListBool :: IO ()
testListBool = let s      = MyState (fromList reqEnv) ""
                   reqEnv = [(("Bool","True")  , ConInfo 1)
                            ,(("Bool","False"), ConInfo 0)
                            ,(("[]",":")      , ConInfo 3)
                            ,(("[]","[]")     , ConInfo 1)]
               in testM (genDefault s :: Gen [Bool]) 10 >>= mapM_ print
```

```
> testListBool
(3,[])
(3,[True])
(1,[True, True])
(1,[True, True, True])
(3,[True, True, True, True])
```

Section 3

Size and order constraints

Size and order constraints: I

Another type of constraint is enforcing *order* on the *size* of a specific argument to a constructor. We will need to carry more state:

```
data MyState = MyState { envState      :: Env
                       , dtNameState   :: String
                       , ctNameState   :: String    -- constructor name
                       , argIndexState :: Int }     -- argument index

type Env     = Map ConLoc ConInfo

type ConLoc  = (String,String)
type ArgLoc  = Int

data ConInfo = ConInfo { freqInfo :: Int
                       , argInfo  :: Map ArgLoc ArgInfo }

data ArgInfo = ArgInfo { ordInfo :: Ordering    -- ordering
                       , sizeInfo :: Int }       -- size
```

Size and order constraints: I

Another type of constraint is enforcing *order* on the *size* of a specific argument to a constructor. We will need to carry more state:

```
data MyState = MyState { envState      :: Env
                       , dtNameState   :: String
                       , ctNameState   :: String   -- constructor name
                       , argIndexState :: Int }    -- argument index
type Env     = Map ConLoc ConInfo
type ConLoc  = (String,String)
type ArgLoc  = Int
data ConInfo = ConInfo { freqInfo :: Int
                       , argInfo  :: Map ArgLoc ArgInfo }
data ArgInfo = ArgInfo { ordInfo :: Ordering   -- ordering
                       , sizeInfo :: Int }     -- size
```

. . . and now we'll have to update the state during generation!

Size and order constraints: II

Our generator now behaves like a state monad:

$\textbf{class}\ \mathit{Generate}\ \alpha\ \textbf{where}$
$\quad \mathit{gen}\ \ :: \mathit{MyState} \rightarrow (\mathit{Gen}\ \alpha, \mathit{MyState})$

$\textbf{class}\ \mathit{GGenerate}\ (\upsilon :: \mathit{Univ})\ \textbf{where}$
$\quad \mathit{gGen} :: \mathit{MyState} \rightarrow (\mathit{Gen}\ (\llbracket\, \upsilon\, \rrbracket), \mathit{MyState})$

At every step, we might change the state, returning an updated version.

Size and order constraints: III

The important changes are in the case for constructor arguments:

```
instance (Elements α, Generate α) ⇒ GGenerate (K ι α) where
  gGen s = let info       = ...   -- lookups
               arg o i    = (elements (elems i), next o i)
               next EQ    = id
               next GT    = pred
               next LT    = succ
           in case info of
               Nothing      → fmapL K (gen s)
               Just (ai, ci) → let (e, i) = arg (ordInfo ai) (sizeInfo ai)
                               in (fmap K e, updSize s i)
```

```
fmapL    :: (α → β) → (α,γ) → (β,γ)
updSize  :: MyState → Int → MyState
elements :: [α] → Gen α
```

Size and order constraints: IV

Generation of elements constrained by size is best done by *enumeration*. Here's a naive generic enumerator:

```
class GElements (υ :: Univ) where
  gelements :: Int → [ ⟦ υ ⟧ ]
instance GElements Unit where
  gelements 0 = [ Unit ]
  gelements _ = []
instance (GElements α, GElements β) ⇒ GElements (α :+: β) where
  gelements n = map Left (gelements n) ++ map Right (gelements n)
instance (Elements α) ⇒ GElements (K ι α) where
  gelements n = map K (elements n)
```

Size and order constraints: V

For products, we consider all possible size distributions:

```
instance (GElements α, GElements β) ⇒ GElements (α :×: β) where
   gelements n = [l :×: r | (a,b) ← split n
                          , l     ← gelements a
                          , r     ← gelements b]

split :: Int → [(Int,Int)]
split n = [(a,n − a) | a ← [0 . . n]]
```

As usual, we provide a user-facing class, and instances for base types:

```
class Elements α where
   elements :: Int → [α]

instance Elements Bool where
   elements 1 = [False, True]
   elements _ = []
```

Size and order constraints: VI

Now we can generate lists of lists of increasing or decreasing length, for example:

```
-- increasing
[[],[False],[True,False],[True,False,False],[True,False,False,False]]
-- decreasing
[[True,True,False,True],[False,False,True],[True,False]]
-- equal
[[False,True],[False,False],[True,False],[True,False],[False,True]]
```

And we can still provide constructor probabilities!

Size and order constraints: VI

Now we can generate lists of lists of increasing or decreasing length, for example:

```
  -- increasing
[[],[False],[True,False],[True,False,False],[True,False,False,False]]
  -- decreasing
[[True,True,False,True],[False,False,True],[True,False]]
  -- equal
[[False,True],[False,False],[True,False],[True,False],[False,True]]
```

And we can still provide constructor probabilities!

Question: how about constraining the order of the arguments themselves, so we can produce *sorted* lists? It's slightly trickier. . .

Section 4

Constructor count constraints

Constructor count constraints: I

Finally, let us try to constrain constructors to occur a *minimum* or *maximum* number of times:

```
data MyState = MyState { envState      :: Env
                       , dtNameState   :: String
                       , ctNameState   :: String
                       , argIndexState :: Int }

type Env     = Map ConLoc ConInfo

data ConInfo = ConInfo { freqInfo :: Int
                       , minInfo  :: Int   -- minimum count
                       , maxInfo  :: Int   -- maximum count
                       , argInfo  :: Map ArgLoc ArgInfo }
```

Constructor count constraints: II

The interesting case is at the sum. We assume right-nesting, so there's one argument to the left of a sum, and (potentially) multiple to the right:

```
instance (GGenerate α, GConNames α
         , GGenerate β, GConNames β) ⇒ GGenerate (α :+: β) where
  gGen s =
    let aNames = ...
        dir = pickLR (dtNameState s) aNames bNames (envState s)
        aFreq = if dir ≡ GoRight then 0 else frequencies ...
        bFreq = if dir ≡ GoLeft  then 0 else frequencies ...
        (genL, sL) = fmapL Left  (gGen s)
        (genR, sR) = fmapL Right (gGen s)
    in (frequency [(aFreq, genL), (bFreq, genR)], s)   -- s?
```

Constructor count constraints: III

For each constructor, we'll check whether we should or shouldn't produce it:

```
data CanGo = Yes | No | DontCare

instance Monoid CanGo where
  mempty = DontCare

  mappend DontCare x        = x
  mappend x        DontCare = x
  mappend No       _        = No
  mappend _        No       = No
  mappend Yes      Yes      = Yes
```

Constructor count constraints: IV

We use this information to decide whether to go left or right:

```
data Dir = GoLeft | GoRight | DoesntMatter
pickLR :: String → [String] → [String] → Env → Dir
pickLR dn cnsL cnsR env =
  let goHere cn = case lookup (dn, cn) env of
                    Nothing → DontCare
                    Just i  → if maxInfo i ≡ 0 then No
                              else if minInfo i > 0 then Yes
                                   else DontCare
  in case (mconcat (map goHere cnsL), mconcat (map goHere cnsR)) of
     (No, No) → DoesntMatter
     (Yes, _) → GoLeft
     (No, _)  → GoRight
     _        → if any (≡ Yes) (map goHere cnsR) then GoRight
                else if all (≡ No) (map goHere cnsR) then GoLeft
                     else DoesntMatter
```

Constructor count constraints: V

Now we can generate lists of length between 3 and 7 without *False*s:

[*True*, *True*, *True*]
[*True*, *True*, *True*, *True*, *True*]
[*True*, *True*, *True*, *True*]

Constructor count constraints: V

Now we can generate lists of length between 3 and 7 without *False*s:

[*True*, *True*, *True*]
[*True*, *True*, *True*, *True*, *True*]
[*True*, *True*, *True*, *True*]

. . . but a number of problems remain:

- We're too eager to reach the requested minimum; if we request lists with a minimum of one [], we only generate empty lists.

Constructor count constraints: V

Now we can generate lists of length between 3 and 7 without *False*s:

[*True*, *True*, *True*]
[*True*, *True*, *True*, *True*, *True*]
[*True*, *True*, *True*, *True*]

. . . but a number of problems remain:

- We're too eager to reach the requested minimum; if we request lists with a minimum of one [], we only generate empty lists.
- It doesn't really work on types like [[*Int*]]

Constructor count constraints: V

Now we can generate lists of length between 3 and 7 without *False*s:

[*True*, *True*, *True*]
[*True*, *True*, *True*, *True*, *True*]
[*True*, *True*, *True*, *True*]

. . . but a number of problems remain:

- We're too eager to reach the requested minimum; if we request lists with a minimum of one [], we only generate empty lists.
- It doesn't really work on types like [[*Int*]]
- . . . or on mutually recursive types in general:

data $A = A_1\ B\ B \mid A_2\ B$
data $B = B_1$

Section 5

Conclusion

Conclusion

We've seen a first attempt at generic generation of constrained random data, with a number of limitations:

- How to distinguish the inner and outer constructors of a type such as $[[\mathit{Int}]]$?

Conclusion

We've seen a first attempt at generic generation of constrained random data, with a number of limitations:

- How to distinguish the inner and outer constructors of a type such as $[[Int]]$?
- Backtracking?

Conclusion

We've seen a first attempt at generic generation of constrained random data, with a number of limitations:

- How to distinguish the inner and outer constructors of a type such as $[[\mathit{Int}]]$?
- Backtracking?
- Generation and enumeration: can they be unified?

Conclusion

We've seen a first attempt at generic generation of constrained random data, with a number of limitations:

- How to distinguish the inner and outer constructors of a type such as $[[\mathit{Int}]]$?
- Backtracking?
- Generation and enumeration: can they be unified?
- Other types of constraints