Building test.check Generators

Gary Fredericks

Who Am I?

- I am Gary Fredericks
- I live in Chicago
- I work (with Clojure) at DRW
- I sometimes work on improving `test.check`

Context

- `test.check`

Context

- `test.check`
  - Created 4 years ago by Reid Draper
  - Clojure library for property-based-testing
    - Haskell's QuickCheck
  - Inputs described by writing generators

Context

- `test.check`
    - Created 4 years ago by Reid Draper
    - Clojure library for property-based-testing
        - Haskell's QuickCheck
    - Inputs described by writing generators
- `clojure.spec`

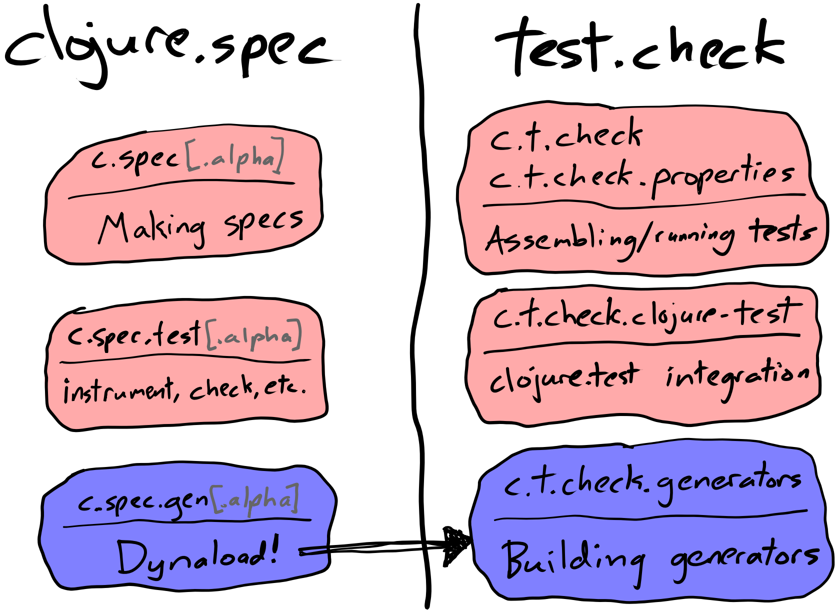
Context

- `test.check`
  - Created 4 years ago by Reid Draper
  - Clojure library for property-based-testing
    - Haskell's QuickCheck
  - Inputs described by writing generators
- `clojure.spec`
  - Announced 18 months ago
  - Creates generates for basic specs
  - User-supplied generators for complex specs

Roadmap

- Generators, in General
- Building
- Fine-Tuning

Generators, in General


clojure.spec
test.check
c.spec[.alpha]
Making specs
c.spec.test[.alpha]
instrument, check, etc.
c.spec.gen[.alpha]
Dynaload!
c.t.check
c.t.check.properties
Assembling/running tests
c.t.check.clojure-test
clojure.test integration
c.t.check.generators
Building generators

Generators, in General

What's in the box?

Basic Data Generators

```clojure
(def generate-some-great-data
  (gen/hash-map
   :a-boolean                gen/boolean
   :some-small-integers      (gen/vector gen/nat)
   :a-large-integer          gen/large-integer
   :a-double                 gen/double
   :a-color                  (gen/elements [:red :green :blue])
   :a-uuid                   gen/uuid
   :a-string-and-a-keyword (gen/tuple gen/string
                                      gen/keyword)))

(gen/generate generate-some-great-data 10)
=>
{:a-boolean               false,
 :some-small-integers     [2 5 5 5 10 5 0],
 :a-large-integer         -6,
 :a-double                0.47607421875,
 :a-color                 :green
 :a-uuid                  #uuid "a06e2893-6fcc-4b42-8e2f-ba5da58202ac",
 :a-string-and-a-keyword ["õ)" :eA:5C*:02]}
```

Combinators!

- `(gen/tuple g1 g2 ...)`
- `(gen/fmap (fn [x] x') g)`
- `(gen/bind g (fn [x] g'))`
- `(gen/such-that pred g)`
- `(gen/frequency [[w1 g1] [w2 g2] ...])`
- `(gen/one-of [g1 g2 ...])`

() -> Data

```
(defn gen-fav-number-assertion
  []
  (let [x (rand-int 10)]
    (str "My favorite number is " x)))

(gen-fav-number-assertion)
=> "My favorite number is 9"
```

Size -> Data

```
(defn gen-fav-number-assertion
  [size]
  (let [x (rand-int size)]
    (str "My favorite number is " x)))

(gen-fav-number-assertion 100)
=> "My favorite number is 91"
```

Size -> (Data, Shrinks Data)

```clojure
(defn gen-fav-number-assertion
  [size]
  (let [x (rand-int size)]
    [(str "My favorite number is " x)
     ;; some sort of recursively lazy
     ;; expression that generates a
     ;; lazy tree of smaller strings
     ]))

(gen-fav-number-assertion 100)
=> ["My favorite number is 91" (...)]
```

Rand Double -> Size -> (Data, Shrinks Data)

```clojure
(defn gen-fav-number-assertion
  [rng size]
  (let [x (-> rng (rand/rand-double) (* size) (long))]
    [(str "My favorite number is " x)
     ;; some sort of recursively lazy
     ;; expression that generates a
     ;; lazy tree of smaller strings
     ]))

(gen-fav-number-assertion (rand/make-random 42) 100)
=> ["My favorite number is 91" (...)]
```

Composition

```clojure
(defn collection-of
  "Returns a generator of a collection
  with elements generated from the
  supplied generator."
  [gen]
  (fn [rng size]
    ;; and now for the tricky bit
    [
     ;; um
     ]))
```

Value Proposition

`clojure.core/rand-int`

vs

`gen/large-integer`

- Abstract sizing/growth
- Shrinking
- Functional Determinism

Dev Tools

```
(gen/sample g <num-samples=10>)
=> (data0 data1 data2 ...)

(gen/generate g <size=30>)
=> data
```

Universality & Purity

- `gen/tuple`
- `gen/one-of`
- `gen/frequency`
- `gen/such-that`
- `gen/fmap`
- `gen/bind`

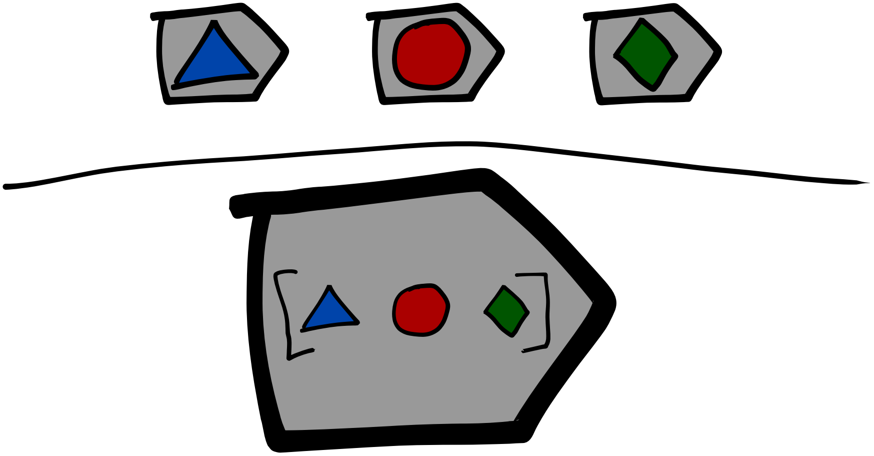
Similarities with FP:

- Universality
  - The built-in generators can generate just about anything
  - But understanding how takes practice
- Circumventability
  - e.g.,
    - `rand`, etc.
    - `gen/sample`, `gen/generate`
    - using today's date, DB records
  - Undermines the value proposition
    - Abstract sizing/growth
    - Shrinking
    - Functional Determinism

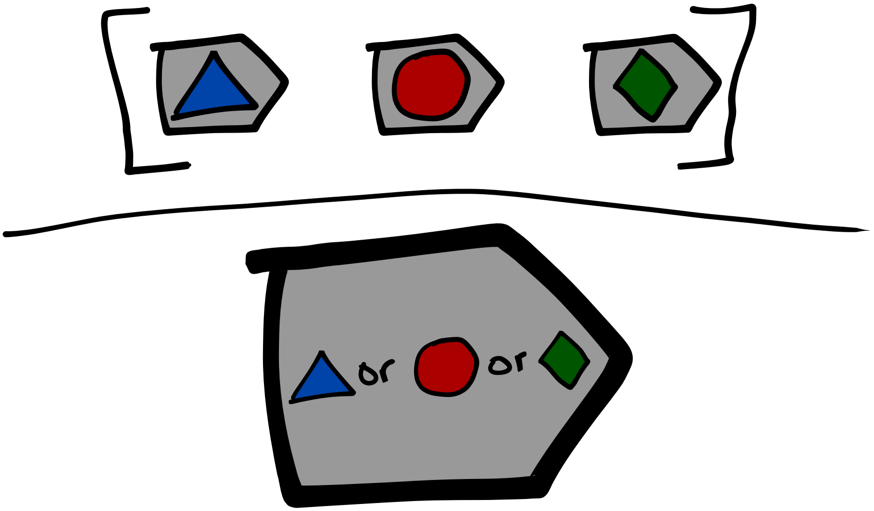
Combinators, this time in detail

- `(gen/tuple g1 g2 ...)`
- `(gen/fmap (fn [x] x') g)`
- `(gen/bind g (fn [x] g'))`
- `(gen/such-that pred g)`
- `(gen/frequency [[w1 g1] [w2 g2] ...])`
- `(gen/one-of [g1 g2 ...])`

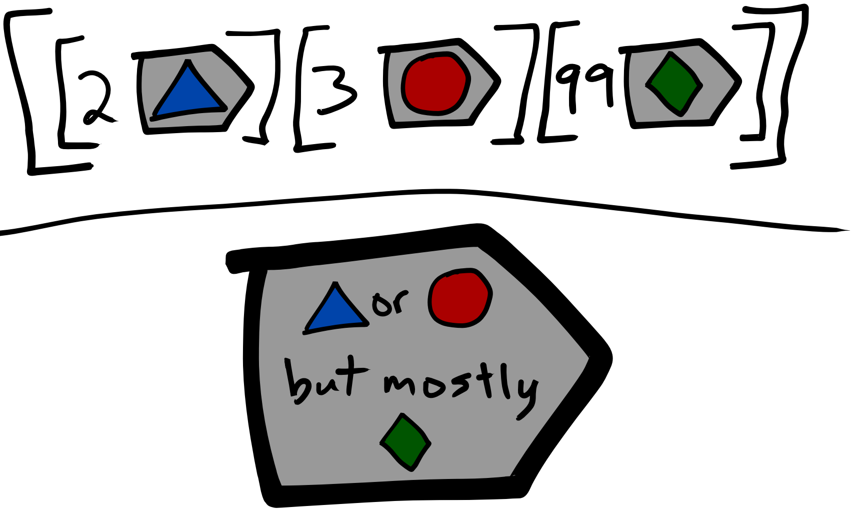
gen/tuple



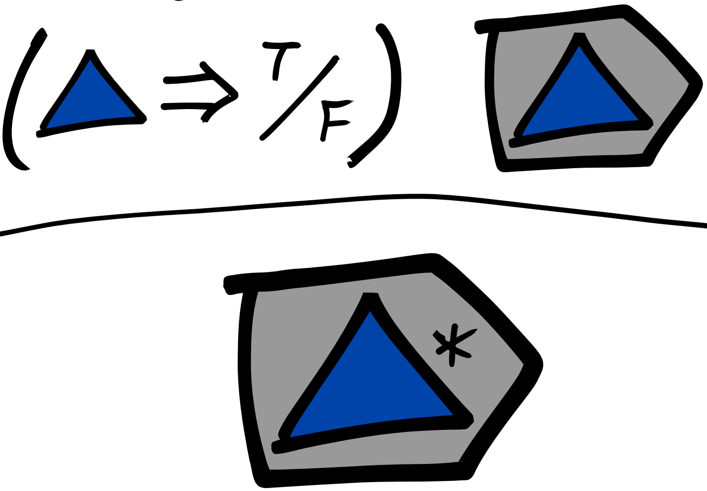
gen/one-of



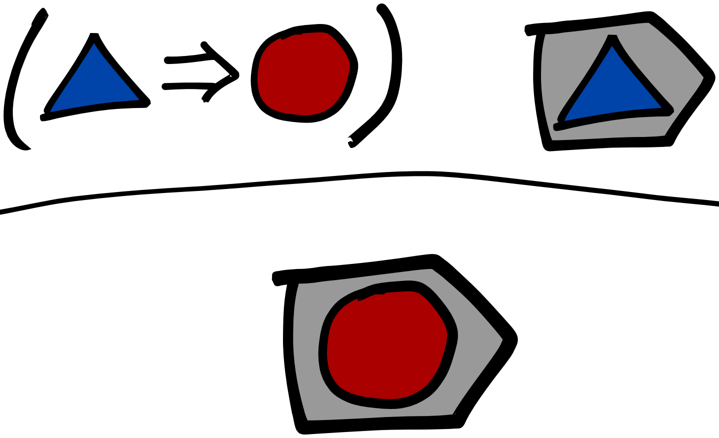
gen/frequency



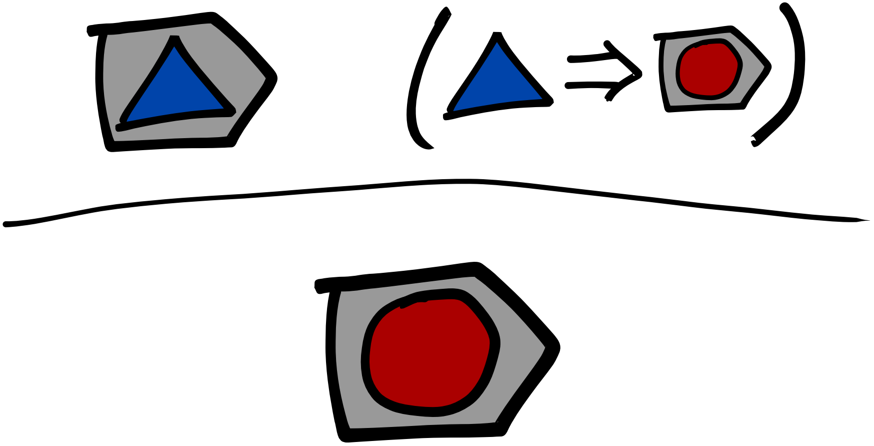
gen/such-that

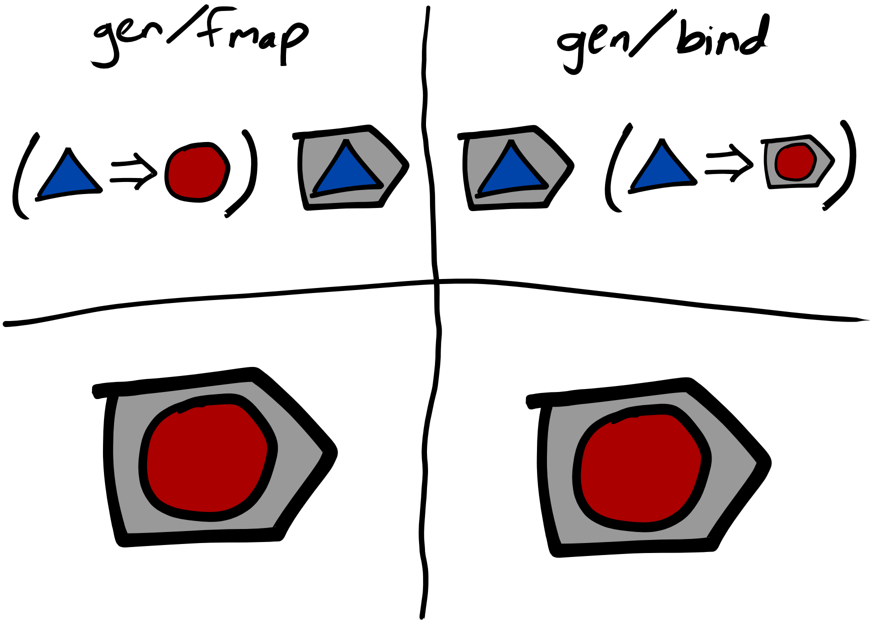


gen/fmap



gen/bind




gen/fmap
gen/bind

Example:

Generate a non-empty collection and a random element from that collection.

Example: collection-and-element

```
(def gen-collection-and-element
  (gen/fmap (fn [coll]
              ;; need to pick a random element
              ;; somehow...
              )
            (gen/not-empty
             (gen/vector gen/large-integer))))
```

Example: collection-and-element

```
(def gen-collection-and-element
  (gen/fmap (fn [coll]
              ;; cheat!
              [coll (rand-nth coll)])
            (gen/not-empty
             (gen/vector gen/large-integer))))

(gen/generate gen-collection-and-element 20)
[[-582 -159 -1 2 -29 -9 -18605 -2 961 -31 90] -9]
```

Example: collection-and-element

```clojure
(def gen-collection-and-element
  (gen/fmap (fn [coll]
              ;; pick a random element, then use
              ;; gen/fmap to wrap it up with the
              ;; collection
              (gen/fmap (fn [x] [coll x])
                        (gen/elements coll)))
            (gen/not-empty
             (gen/vector gen/large-integer))))

(gen/generate gen-collection-and-element)
#clojure.test.check.generators.Generator{:gen #function[clo
;; we just generated a generator, not data
```

Example: collection-and-element

```clojure
(def gen-collection-and-element
  (gen/bind (gen/not-empty
             (gen/vector gen/large-integer))
            (fn [coll]
              ;; pick a random element, then use
              ;; gen/fmap to wrap it up with the
              ;; collection
              (gen/fmap (fn [x] [coll x])
                        (gen/elements coll)))))

(gen/generate gen-collection-and-element 20)
[[-1 40 3 6199 -77 -433763 -412 46055 0 -6 0] 40]
```

Example: collection-and-element

```
(def gen-collection-and-element
  (gen/let [coll (gen/not-empty
                  (gen/vector gen/large-integer))
            x    (gen/elements coll)]
    [coll x]))

(gen/generate gen-collection-and-element 20)
[[-37830 3546 -210996 3 0 -35206 6 517097] 3546]
```

Example: a 2d matrix

```clojure
;; Bad, because the inner vectors
;; can have different lengths.
(gen/vector (gen/vector gen/large-integer))


;; Generate a width first, then use bind to generate
;; a collection of vectors with the same width.
(gen/let [width gen/nat
          rows (gen/vector
                (gen/vector gen/large-integer
                            width))]
  rows)
```

Summary: Generators, in General

- There are concrete data generators, and abstract combinators
- The abstractness is in service of the value proposition
- Like FP - familiarity takes practice

Building

Long-Running Example: Dropbox

- Generate a random (nested) directory of files
- Generate changes to a directory

gen-file-name

```
(def gen-file-name
  (gen/such-that #(not (re-find #"/" %))
                 gen/string-ascii))

(gen/sample gen-file-name)
("" "" "F" "kI$" "0%" "T4\\:" "W`\\:P" "qS4" "" "")
```

gen-file-name

```
(gen/sample gen-file-name 1000)
;; ExceptionInfo Couldn't satisfy such-that
;; predicate after 10 tries.
;; clojure.core/ex-info (core.clj:4725)
```

Common culprit in spec: `(s/and s1 s2 s3 ...)`

gen-file-name-2

```clojure
(def gen-file-name-2
  (gen/fmap #(clojure.string/replace % "/" "")
            gen/string-ascii))

(gen/sample gen-file-name-2)
("" "O" "E" "=e+" "ozdH"
 "_" "WNL_6" "{Zqm" "CY(1I)H." "z")
```

gen-file-contents

```
(def gen-file-contents
  gen/bytes)

(gen/sample gen-file-contents)
(#bytes ""
 #bytes "0e"
 #bytes "8b"
 #bytes "80e1d5"
 #bytes "8ad6"
 #bytes ""
 #bytes "b0a0224119c8"
 #bytes "f39a97ff"
 #bytes "8a"
 #bytes "22908190fc09eca901")
```

gen-file-metadata

```
(def gen-permissions-octal
  (gen/fmap #(format "%03o" %)
            (gen/large-integer {:min 0 :max 0777})))

(gen/sample gen-permissions-octal)
("350" "330" "350" "017" "150"
 "155" "666" "105" "533" "634")
```

gen-file-metadata

```
(def gen-datetime
  (gen/fmap #(java.time.Instant/ofEpochMilli %)
            gen/large-integer))

(gen/sample gen-datetime)
(#inst "1970-01-01T00:00:00.000Z"
 #inst "1969-12-31T23:59:59.999Z"
 #inst "1969-12-31T23:59:59.998Z"
 #inst "1970-01-01T00:00:00.000Z"
 #inst "1969-12-31T23:59:59.998Z"
 #inst "1970-01-01T00:00:00.007Z"
 #inst "1970-01-01T00:00:00.000Z"
 #inst "1969-12-31T23:59:59.938Z"
 #inst "1969-12-31T23:59:59.999Z"
 #inst "1970-01-01T00:00:00.035Z")
```

gen-file-metadata

```clojure
(def gen-metadata
  (gen/hash-map :permissions gen-permissions-octal
                :user-id     gen/large-integer
                :group-id    gen/large-integer
                :created-at  gen-datetime
                :modified-at gen-datetime))

(gen/generate gen-metadata)
{:permissions "161",
 :user-id     484817,
 :group-id    10350453,
 :created-at  #inst "1970-01-01T00:01:03.269Z",
 :modified-at #inst "1970-01-01T00:00:00.009Z"}
```

gen-directory

```clojure
(defn gen-directory-of
  [gen-content]
  (gen/map gen-file-name
           (gen/hash-map :metadata gen-metadata
                         :content  gen-content)))

(def gen-directory
  ;; use gen/such-that to filter out
  ;; top-level byte arrays
  (gen/such-that map?
                 (gen/recursive-gen
                  gen-directory-of
                  gen-file-contents)))
```

gen-directory

```
(gen/generate gen-directory 10)

{"eqn" {:metadata {:created-at #inst "1970-01-01T00:00:00.003Z",
                   :group-id 0,
                   :modified-at #inst "1969-12-31T23:59:59.999Z",
                   :permissions "045",
                   :user-id 0},
        :content {")" {:metadata {:created-at #inst "1969-12-31T23:59:5
                                  :group-id 0,
                                  :modified-at #inst "1970-01-01T00:00
                                  :permissions "760",
                                  :user-id 0},
                       :content #bytes "c5a1159d"}}},
 "xu_" {:metadata {:created-at #inst "1970-01-01T00:00:00Z",
                   :group-id 0,
                   :modified-at #inst "1969-12-31T23:59:59.999Z",
                   :permissions "372",
                   :user-id 0},
        :content {"M" {:metadata {:created-at #inst "1969-12-31T23:59:5
                                  :group-id -1,
                                  :modified-at #inst "1970-01-01T00:00
```

gen-directory

```clojure
(def a-good-directory *1)
```

gen-changes

```
(def gen-directory-with-changes
  ????)
```

gen-changes

```clojure
(defn gen-changes
  [directory]
  ????)

(def gen-directory-with-changes
  ????)
```

gen-changes

```clojure
(defn gen-changes
  [directory]
  ????)

(def gen-directory-with-changes
  (gen/bind gen-directory
            (fn [directory]
              (gen/fmap (fn [changes]
                          {:directory directory
                           :changes   changes})
                        (gen-changes directory)))))
```

gen-changes

```clojure
(def gen-directory-with-changes
  (gen/bind gen-directory
            (fn [directory]
              (gen/fmap (fn [changes]
                          {:directory directory
                           :changes   changes})
                        (gen-changes directory)))))
;; same as
(def gen-directory-with-changes
  (gen/let [directory gen-directory
            changes   (gen-changes directory)]
    {:directory directory
     :changes   changes}))
```

gen-changes

```clojure
(defn gen-changes
  [directory]
  (gen/vector (gen-appension directory)))
```

gen-changes

```clojure
(defn gen-appension
  [directory]
  ;; 1. get all filepaths
  ;; 2. select (generate) a filepath
  ;; 3. generate random bytes to append
  ;; 4. package together and return
  )
```

gen-changes

```clojure
(defn all-filepaths
  [directory]
  (apply concat
         (for [[name {:keys [content]}] directory]
           (if (map? content)
             (map #(str name "/" %)
                  (all-filepaths content))
             [name]))))

(all-filepaths a-good-directory)
("eqn/)" "xu_/M" "xu_/X")
```

gen-changes

```clojure
(defn gen-file-appension
  [directory]
  (let [filepaths (all-filepaths directory)]
    (assert (not-empty filepaths))
    (gen/fmap (fn [[filepath more-bytes]]
                {:filepath filepath
                 :append   more-bytes})
              (gen/tuple (gen/elements filepaths)
                         gen/bytes))))

(gen/generate
 (gen-file-appension a-good-directory))
{:filepath "xu_/X",
 :append #bytes "3b5cfeb016458d"}
```

gen-changes

All together now. . .

gen-changes

```
(defn gen-changes
  [directory]
  ;; do something different if it's empty
  (gen/vector (gen-file-appension directory)))

(def gen-directory-with-changes
  (gen/let [directory gen-directory
            changes   (gen-changes directory)]
    {:directory directory
     :changes   changes}))

(gen/generate gen-directory-with-changes 3)
{:directory {"" {:metadata {:permissions "217",
                            :user-id -1,
                            :group-id 0,
                            :created-at #inst "1970-01-01T00:00:00.000
                            :modified-at #inst "1969-12-31T23:59:59.99
                 :content #bytes ""}},
 :changes [{:filepath "", :append #bytes ""}]}
```

Summary: Building Things

Just like FP, building generators takes practice

Fine-Tuning

Fine-tune what, exactly?

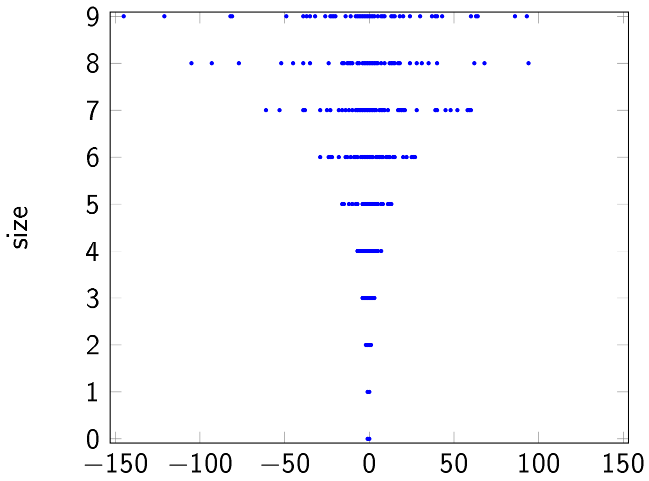
Fine-tune what, exactly?

- Distribution/Growth
- Shrinking

Fine-Tuning

Distribution/Growth

gen/large-integer



Distribution of `gen/large-integer` for `size=0..9`

What does `size` mean exactly?

Its meaning is determined by its use:

- Test runs use `0,1,2,...,199,0,1,2,...`

size gotchas

- Testing with `:num-tests < 200`
- `(gen/sample g)` is `size=0..9`
- `(gen/generate g)` is `size=30`

gen/scale

"Big" is relative.

string vectors

```clojure
(def g (gen/vector gen/string-ascii))

(repeatedly 5 #(gen/generate g 10))

(["V5N8`!)/" "_{^)X2ml"]
 ["<h}>YmK^$2" "" "" "B-c>M/" ""]
 ["b.e2I*$76"]
 ["Vo" "Sk"]
 ["u"])
```

string vectors

```
(gen/generate g 200)

["s9&\"(5y?%{X6AuFU2dkM\"m0kb+1c(-mb@]C)n.$)g|/joLvU~W,g~L"
 "&_<#bCsH7A-~\"_V=(r(i|G-'9bF.}N*NYass;*sWt(g06Y7XRZ37)6aQ:)lm~xgq(%xKIuZ>2V<+4dxsW7[eNygK!0b{y)R/o>()E
 ".{?T-@psGc%VDrQ{R;>5u0*/0dWfN=)\\;)3H}Ni>V0Y!R63C@[^LdVZ?W0nAs|Hem0v0("
 "$~M(2ppm`H|*x$Z,0m39<2WbXbwbib=_V2o^Jv-yT~~7BRci]$?vc8/l)"
 "u~O>dYQG7>849<%DHTlliFUk>TcUMJ`hkk*\"M\\KXl.shq8ExD=KsQ7J(mI{i5Mh8<wG`n{|eL9e%c'K){,*PtYDW|pKU?,uouDsK
 "GoP>[iISnY*wEGjBB^)+@(GW1$]h%aR.{JYMKq*X .]C6ty8e#%Y9.e:>GlE(RN;f2A\\5/om?o)%"
 "h1=Z;6cJed`c\"%_G+H.qg?@oS>"
 "x[I}i1K+te6-}KTqQU<cl"
 "%5$\\u(i,>uwWz'evt^Q?=m%H/t$~UB+_GpCdrTV?g3YJM2u$C,&1f.2Ym\\Q 'l8H>]a35FA4%Ih,\\e~Bv\\!Qfr6Kc_;`>T5`9D
 "&_w5s69S%M:ZWU^q~hoYva%_$hDiraW~auqO>P#G-].z 55w&"
 "'-OM=-uP];m8U?Q2(PsuPEp{F-NV'liq_\\,[K8.ABaTQI'MoqbU' _jGXz!$t$qm}!j~V!l'Y+GU6*q7b7v-y\\MR,*"
 "i<d-b0_g(m~iMkLq4]@#srM9C(g&`J%bz.sA+BNX~0Cj\"J&}&]).oMFA.%-tyBG]bCZ%h1D,]*~R~{2C,"
 "pl]pVTI wVD4u]t)1,Et&)r9Tj,: 7+XzC8gRhWe8Ioc41102s?+Ueew>U{U(lx\"c<<1hSTs3ak.LbX3g}{;sI6w_`,C0l*s)rbA:
 "2$XMP\\\"GE;xTkLBu}RX@LQ S/S7[ziMI%GE0ZY]I`drF<5oDTi5&EQgN;l0Ne}}!c2^90<JRl<%T_ArU1voHI\\$%;"
 "LYmx,_u;,&n4%:>U-q>WCl]WGg)06}w"
 "H\"EH%MpHD6UP,_@v04kd9rDeK]Tk!H]M5=xSv3(e}_W3Y30R%zkI'^GFgXaR.YIU00UW'-CjVZQ.Duw3$8=;tq#Tje.Y"
 "AU(%)Be>)WJZ4q)4Q:P?Z?=!Mh>[>>wfK7q&H"
 "Zpw|d)R|{C\\e&XfN*|,K@|. N.6W?N'>p5N:@>UBLx4h,G5xRBPM^!Fz^zm]zMV[ufJg.\"6wLjBr{#`5t)1zkbbHk=!C$&5B-m<g
 "A~'0h~?F@LG/}+-l\\ds1+X{Inofdj:z=+9S+/SuaLZM<rXf&u~Z>-=7YJ94Zzy[:.e;uDIsyDIsr4~iB(rxi<#E)Ck[H*C_tEIYW*
 ">r.=_3&ElZ.<T(hW\"2L%UG:ur=EmWA;60wE2zPbfWl9|S(*!E=8K5snV!0>T='$=mci&t&!9T.WK[z9\"`X~.`VHyW!=D!XhC~VF6
 "{Z|np!C|F.b*H+t'?wgDefL/<=YHsRt<U=)vX{bk@NSR[UWJx&%J':uY4tu![]FIB8s6)9Pr@[*Y9h8st7z}\"pQ%7s,V]6{/bZM)M
 "Z\"5.e#Cew{k5>^vm?YyUSDTL6;.^^g)Qx(J2I$E5.g@!vM|->3 4\"cE)?LNW604Q>5P}m&A#."
 "* B~NaUS/DhC7Z&A</:7cm\"A>&V7$WX&=\\v&-{Vn=va?[>Z(%o N%C)ta7%/_,lU3dn{a8.SP\\8`/t"
 "iTMn}DK\"l]d6Hk6nVn2U[}+bJ6Tk?sG2q&*%GcGDTjJ{v//9TUTj[3\\V:`R:=)8Tguw|<}#-}Ja2f7M=4CQCC>9,q<h76n+:Y43,
```

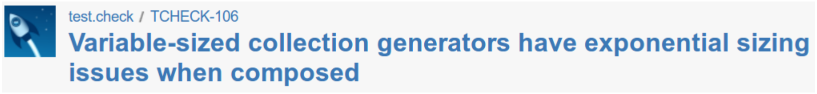
string vectors

```clojure
(def g-2 (gen/scale #(/ % 20)
                    (gen/vector gen/string-ascii)))

(repeatedly 5 #(gen/generate g-2 200))

(["Qw+b"]
 [".S." "C’NL.k2"]
 []
 ["Sh?7?Wv" "" "@,~d_^*z" "z<" "]Q!" "XPVS"
  "/|@)WYvbH" "(=jTr" "ZQ}k[as;oO" "p&ri~;f1Q6"]
 ["O=]{-=\\" "BTqjqWg" ":N1z" "F6I*(P"])
```

PSA


test.check / TCHECK-106

Variable-sized collection generators have exponential sizing issues when composed

Too Weird / Not Weird Enough

Too Weird / Not Weird Enough

```clojure
;; doesn't often generate
;; nontrivial text files
(def gen-file-contents
  gen/bytes)
```

What's the best amount of weird?

Too Weird / Not Weird Enough

```clojure
;; doesn't often generate
;; nontrivial text files
(def gen-file-contents
  gen/bytes)

;; half random bytes, half UTF-8
(def gen-file-contents-2
  (gen/one-of [gen/bytes
               gen-UTF8-bytes]))
```

Too Weird / Not Weird Enough

```clojure
;; Only generates small files
(def gen-file-contents-2
  (gen/one-of [gen/bytes
               gen-UTF8-bytes]))
```

Too Weird / Not Weird Enough

```clojure
;; Only generates small files
(def gen-file-contents-2
  (gen/one-of [gen/bytes
               gen-UTF8-bytes]))

;; Occasionally generates big files!
(def gen-file-contents-3
  (gen/frequency [[45 gen/bytes]
                  [45 gen-UTF8-bytes]
                  [10 (gen/scale
                       (fn [size]
                         (if (<= 100 size)
                           (* size size)
                           size))
                       gen/bytes)]]))
```

gen-datetime

```
(def gen-datetime
  (gen/fmap #(java.time.Instant/ofEpochMilli %)
            gen/large-integer))

(gen/sample gen-datetime)
(#inst "1970-01-01T00:00:00.000Z"
 #inst "1969-12-31T23:59:59.999Z"
 #inst "1969-12-31T23:59:59.998Z"
 #inst "1970-01-01T00:00:00.000Z"
 #inst "1969-12-31T23:59:59.998Z"
 #inst "1970-01-01T00:00:00.007Z"
 #inst "1970-01-01T00:00:00.000Z"
 #inst "1969-12-31T23:59:59.938Z"
 #inst "1969-12-31T23:59:59.999Z"
 #inst "1970-01-01T00:00:00.035Z")
```

gen-datetime-2

```clojure
(def gen-datetime-components
  (gen/hash-map
   :year   (gen/fmap #(+ % 2017) gen/int)
   :month  (gen/large-integer* {:min 1, :max 12})
   :day    (gen/large-integer* {:min 1, :max 31})
   :hour   (gen/large-integer* {:min 0, :max 23})
   :minute (gen/large-integer* {:min 0, :max 59})
   :second (gen/large-integer* {:min 0, :max 59})
   :millis (gen/large-integer* {:min 0, :max 1000})))
```

gen-datetime-2

```clojure
(defn construct-datetime
  [{:keys [year month day
           hour minute second millis]}]
  (try
    (java.time.Instant/parse
     (format "%04d-%02d-%02dT%02d:%02d:%02d.%03dZ"
             year month day
             hour minute second millis))
    (catch Exception e
      ;; kind of dumb, but it works and it's easy
      (java.time.Instant/parse
       (format "%04d-%02d-%02dT%02d:%02d:%02d.%03dZ"
               year month 28
               hour minute second millis)))))
```

gen-datetime-2

```clojure
(def gen-datetime-2
  (gen/fmap construct-datetime
            gen-datetime-components))

(gen/sample gen-datetime-2)
(#inst "2017-02-02T01:01:01.001Z"
 #inst "2016-01-02T01:01:01.000Z"
 #inst "2017-02-01T00:01:00.001Z"
 #inst "2015-02-01T00:03:01.003Z"
 #inst "2016-07-02T01:00:00.002Z"
 #inst "2022-03-02T01:02:03.001Z"
 #inst "2020-05-03T04:00:01.000Z"
 #inst "2019-02-09T10:05:11.000Z"
 #inst "2020-04-08T12:27:04.053Z"
 #inst "2013-08-06T21:04:03.001Z")
```

Fine-Tuning

Shrinking

`gen-datetime` vs `gen-datetime-2`

gen/uuid

```clojure
(def ^{:added "0.9.0"} uuid
  "Generates a random type-4 UUID. Does not shrink."
  (no-shrink
   ...))
```

gen/bind

```clojure
(def gen-matrix
  (gen/let [width gen/nat
            rows (gen/vector
                   (gen/vector gen/large-integer
                               width))]
    rows))

(quick-check 10000
             (prop/for-all [matrix gen-matrix]
               (->> matrix
                    (apply concat)
                    (not-any? #{42}))))

;; fails on a large matrix, shrinks to [[0 42 0]]
```

Custom Shrinking

Summary: Tweaking Things

- Things can be too big or small; `gen/scale` can help
- You can target specific sensitivities with `gen/one-of` and `gen/frequency`
- Modeling the domain better can help
- Shrinking is hard

Welp!

That's About It

- Generator combinators are abstract and declarative, to support growth and shrinking
- With practice, you can generate anything, and customize its distribution, growth, and shrinking

Thanks!