Package ‘hedgehog’

August 22, 2018

Type Package
Title Property-Based Testing
Version 0.1
Date 2018-08-14

Description Hedgehog will eat all your bugs.
'Hedgehog' is a property-based testing package in the spirit of 'QuickCheck'. With 'Hedgehog', one can test properties of their programs against randomly generated input, providing far superior test coverage compared to unit testing. One of the key benefits of 'Hedgehog' is integrated shrinking of counterexamples, which allows one to quickly find the cause of bugs, given salient examples when incorrect behaviour occurs.

License MIT + file LICENSE

URL https://hedgehog.qa

BugReports https://github.com/hedgehogqa/r-hedgehog/issues

Imports rlang (>= 0.1.6)
Depends testthat
Suggests knitr, rmarkdown
VignetteBuilder knitr
RoxygenNote 6.0.1

NeedsCompilation no

Author Huw Campbell [aut, cre]
Maintainer Huw Campbell <huw.campbell@gmail.com>
Repository CRAN

Date/Publication 2018-08-22 16:50:03 UTC
### Topics Documented:

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>2</td>
</tr>
<tr>
<td>discard</td>
<td>3</td>
</tr>
<tr>
<td>expect_sequential</td>
<td>4</td>
</tr>
<tr>
<td>forall</td>
<td>4</td>
</tr>
<tr>
<td>gen-element</td>
<td>5</td>
</tr>
<tr>
<td>gen-monad</td>
<td>7</td>
</tr>
<tr>
<td>gen-actions</td>
<td>8</td>
</tr>
<tr>
<td>gen.beta</td>
<td>9</td>
</tr>
<tr>
<td>gen.c</td>
<td>9</td>
</tr>
<tr>
<td>gen.date</td>
<td>10</td>
</tr>
<tr>
<td>gen.example</td>
<td>10</td>
</tr>
<tr>
<td>gen.gamma</td>
<td>11</td>
</tr>
<tr>
<td>gen.list</td>
<td>11</td>
</tr>
<tr>
<td>gen.no.shrink</td>
<td>12</td>
</tr>
<tr>
<td>gen.recursive</td>
<td>12</td>
</tr>
<tr>
<td>gen.run</td>
<td>13</td>
</tr>
<tr>
<td>gen.shrink</td>
<td>13</td>
</tr>
<tr>
<td>gen.sized</td>
<td>14</td>
</tr>
<tr>
<td>gen.structure</td>
<td>14</td>
</tr>
<tr>
<td>gen.unif</td>
<td>15</td>
</tr>
<tr>
<td>generate</td>
<td>15</td>
</tr>
<tr>
<td>hedgehog</td>
<td>16</td>
</tr>
<tr>
<td>shrink.halves</td>
<td>17</td>
</tr>
<tr>
<td>shrink.list</td>
<td>17</td>
</tr>
<tr>
<td>shrink.removes</td>
<td>18</td>
</tr>
<tr>
<td>shrink.towards</td>
<td>18</td>
</tr>
<tr>
<td>symbolic</td>
<td>19</td>
</tr>
<tr>
<td>tree</td>
<td>19</td>
</tr>
<tr>
<td>tree.replicate</td>
<td>20</td>
</tr>
</tbody>
</table>

### Command

This helper function assists one in creating commands for state machine testing in hedgehog.

### Usage

```r
command(title, generator, execute, require = function(state, ...) T,
  update = function(state, output, ...) state, ensure = function(state,
  output, ...) NULL)
```
discard

Arguments

- **title**: the name of this command, to be shown when reporting any failing test cases.
- **generator**: A generator which provides random arguments for the command, given the current (symbolic) state. If nothing can be done with the current state, one should preclude the situation with a requires and return NULL. Otherwise, it should be a list of arguments (the empty list is ok for functions which take no arguments).
- **execute**: A function from the concrete input, which executes the true function and returns concrete output. Function takes the (possibly named) arguments given by the generator.
- **require**: A function from the current (symbolic) state to a bool, indicating if action is currently applicable. Function also takes the (possibly named) arguments given by the generator (this is mostly used in shrinking, to ensure after a shrink its still something which could have been generated by the function generator).
- **update**: A function from state to state, which is polymorphic over symbolic and concrete inputs and outputs (as it is used in both action generation and command execution). It’s critical that one doesn’t “inspect” the output and input values when writing this function.
- **ensure**: A post-condition for a command that must be verified for the command to be considered a success. This should be a set of testthat expectations.

Value

- a command structure.

---

**discard**  
*Discard a test case*

Description

Discard a test case

Usage

discard()
expect_sequential  Execute a state machine model

Description
Executes the list of commands sequentially, ensuring that all postconditions hold.

Usage
expect_sequential(initial.state, actions)

Arguments
initial.state  the starting state to build from which is appropriate for this state machine generator.
actions      the list of actions which are to be run.

Value
an expectation.

forall  Hedgehog property test

Description
Check a property holds for all generated values.

Usage
forall(generator, property, tests = getOption("hedgehog.tests", 100),
size.limit = getOption("hedgehog.size", 50),
shrink.limit = getOption("hedgehog.shrinks", 100),
discard.limit = getOption("hedgehog.discards", 100),
curry = identical(class(generator), "list"))

Arguments
generator a generator or list of generators (potentially nested) to use for value testing.
property   a function which takes a value from from the generator and tests some predicated against it.
tests      the number of tests to run
size.limit the max size used for the generators
shrink.limit the maximum number of shrinks to run when shrinking a value to find the smallest counterexample.
discard.limit  the maximum number of discards to permit when running the property.
curry    whether to curry the arguments passed to the property, and use do.call to use
          the list generated as individual arguments. When curry is on, the function arity
          should be the same as the length of the generated list. Defaults to T if the input
          is a list.

Details

The generator used can be defined flexibly, in that one can pass in a list of generators, or even nest
generators and constant values deeply into the gen argument and the whole construct will be treated
as a generator.

Examples

test_that("Reverse and concatenate symmetry",
  forall( list( as = gen.c( gen.element(1:100) )
    , bs = gen.c( gen.element(1:100) )
    , function( as, bs )
      expect_identical( rev(c(as, bs)), c(rev(bs), rev(as)))
    )
  )
)

# False example showing minimum shrink:
## Not run:
test_that("Reverse is identity",
  forall( gen.c( gen.element(1:100) ), function(x) { expect_identical( rev(x), c(x) ) } )
)

## End(Not run)
# Falsifiable after 1 tests, and 5 shrinks
# Predicate is falsifiable

# Counterexample:
# [1] 1 2

---

Random Sample Generation

Description

Generators which sample from a list or produce random integer samples. Both single sample, with
gen.element; and multi-sample, with gen.sample and gen.subsequence are supported; while
gen.choice is used to choose from generators instead of examples.
Usage

- gen.element(x, prob = NULL)
- gen.int(n, prob = NULL)
- gen.choice(..., prob = NULL)
- gen.subsequence(x)
- gen.sample(x, size, replace = FALSE, prob = NULL)
- gen.sample.int(n, size, replace = FALSE, prob = NULL)

Arguments

- x: a list or vector to sample an element from.
- prob: a vector of probability weights for obtaining the elements of the vector being sampled.
- n: the number which is the maximum integer sampled from.
- ...: generators to sample from
- size: a non-negative integer or a generator of one, giving the number of items to choose.
- replace: Should sampling be with replacement?

Details

These generators implement shrinking.

Value

- gen.element returns an item from the list or vector; gen.int, an integer up to the value n; gen.choice, a value from one of given selected generators; gen.subsequence an ordered subsequence from the input sequence; and gen.sample a list or vector (depending on the input) of the inputs.

For gen.element and gen.choice, shrinking will move towards the first item; gen.int will shrink to 1; gen.subsequence will shrink the list towards being empty; and gen.sample will shrink towards the original list order.

Examples

- gen.element(1:10) # a number
- gen.element(c(TRUE,FALSE)) # a boolean
- gen.int(10) # a number up to 10
- gen.choice(gen.element(1:10), gen.element(letters))
- gen.choice(NaN, Inf, gen.unif(-10, 10), prob = c(1,1,10))
- gen.subsequence(1:10)
**Description**

A Hedgehog generator is a function, which, using R’s random seed, will build a lazy rose tree given a size parameter, which represent a value to test, as well as possible shrinks to try in the event of a failure. Usually, one should compose the provided generators instead of dealing with the gen constructor itself.

**Usage**

- `gen(t)`
- `gen.and_then(g, f)`
- `gen.bind(f, g)`
- `gen.pure(x)`
- `gen.impure(fg)`
- `gen.with(g, m)`
- `gen.map(m, g)`

**Arguments**

- `t` a function producing a tree from a size parameter, usually an R function producing random values is used.
- `g` a generator to map or bind over
- `f` a function from a value to new generator, used to build new generators monadically from a generator’s output
- `x` a value to use as a generator
- `fg` a function producing a single value from a size parameter
- `m` a function to apply to values produced the generator

**Details**

Hedgehog generators are functors and monads, allowing one to map over them and use their results to create more complex generators.

A generator can use R’s random seed when constructing its value, but all shrinks should be deterministic.

In general, functions which accept a generator can also be provided with a list of generators nested arbitrarily.
Generators which are created from impure values (i.e., have randomness), can be created with `gen.impure`, which takes a function from size to a value. When using this the function will not shrink, so it is best composed with `gen.shrink`.

See Also

`generate` for way an alternative, but equally expressive way to compose generators using R’s “for” loop.

Examples

```r
# Create a generator which produces a number between 1 and 30
one_to_30 <- gen.element(1:30)

# Use this to create a simple vector of 6 numbers between 1 and 30.
vector_one_to_30 <- gen.c(of = 6, one_to_30)

# Create a matrix 2 by 3 matrix using said vector
gen.map(function(x) matrix(x, ncol=3), vector_one_to_30)

# To create a generator from a normal R random function
# use gen.impure (this generator does not shrink).
g <- gen.impure(function(size) sample(1:10))
gen.example(g)
# [1]  5  6  3  4  8 10  2  7  9  1

# Composing generators with `gen.bind` and `gen.with` is easy. Here we make a generator which first build a length,
# then, elements of that length.
g <- gen.bind(function(x) gen.c(of = x, gen.element(1:10)), gen.element(2:100))
gen.example(g)
# [1]  8  6  2  7  5  4  2  4  2  4  6  4  6  6  3  6  7  8  5  4  6
```

---

`gen.actions`  
Generate a list of possible actions.

Description

Generate a list of possible actions.

Usage

`gen.actions(initial.state, commands)`
**Arguments**

- **initial.state**
  - the starting state to build from which is appropriate for this state machine generator.

- **commands**
  - the list of commands which we can select choose from. Only commands appropriate for the state will actually be selected.

**Value**

- a list of actions to run during testing

---

**Description**

Shrinks towards the median value.

**Usage**

```
gen.beta(shape1, shape2, ncp = 0)
```

**Arguments**

- **shape1**
  - same as shape1 in rbeta

- **shape2**
  - same as shape2 in rbeta

- **ncp**
  - same as ncp in rbeta

---

**Description**

Generate a vector of values from a generator

**Usage**

```
gen.c(generator, from = 1, to = NULL, of = NULL)
```

**Arguments**

- **generator**
  - a generator used for vector elements

- **from**
  - minimum length of the list of elements

- **to**
  - maximum length of the list of elements (defaults to size if NULL)

- **of**
  - the exact length of the list of elements (exclusive to ‘from’ and ‘to’).
**gen.date**

Generate a date between the from and to dates specified.

**Description**

Shrinks towards the from value.

**Usage**

```r
gen.date(from = as.Date("1990-01-01"), to = as.Date("3000-01-01"))
```

**Arguments**

- `from` a Date value
- `to` a Date value

**Examples**

```r
gen.date()
gen.date(from = as.Date("1939-09-01"), to = as.Date("1945-09-02"))
```

---

**gen.example**

Sample from a generator.

**Description**

Sample from a generator.

**Usage**

```r
gen.example(g, size = 5)
```

**Arguments**

- `g` A generator
- `size` The sized example to view
**gen.gamma**

*Generate a float with a gamma distribution*

**Description**

Shrinks towards the median value.

**Usage**

```r
gen.gamma(shape, rate = 1, scale = 1/rate)
```

**Arguments**

- **shape**: same as shape in rgamma
- **rate**: same as rate in rgamma
- **scale**: same as scale in rgamma

---

**gen.list**

*Generate a list of values, with length bounded by the size parameter.*

**Description**

Generate a list of values, with length bounded by the size parameter.

**Usage**

```r
gen.list(generator, from = 1, to = NULL, of = NULL)
```

**Arguments**

- **generator**: a generator used for list elements
- **from**: minimum length of the list of elements
- **to**: maximum length of the list of elements (defaults to size if NULL)
- **of**: the exact length of the list of elements (exclusive to ‘from’ and ‘to’).
**gen.no.shrink**  
*Stop a generator from shrinking*

**Description**  
Stop a generator from shrinking

**Usage**  
`gen.no.shrink(g)`

**Arguments**  
`g` a generator we wish to remove shrinking from

**gen.recursive**  
*Build recursive structures in a way that guarantees termination.*

**Description**  
This will choose between the recursive and non-recursive terms, while shrinking the size of the recursive calls.

**Usage**  
`gen.recursive(tails, heads)`

**Arguments**  
`tails` a list of generators which should not contain recursive terms.
`heads` a list of generator which may contain recursive terms.

**Examples**
```r
# Generating a tree with integer leaves
treeGen <-
gen.recursive(
  # The non-recursive cases
  list(
    gen.int(100)
  ),
  # The recursive cases
  list(
    gen.list( treeGen )
  )
)
```
**gen.run**

*Run a generator*

**Description**

Samples from a generator or list of generators producing a (single) lazy rose tree.

**Usage**

```
gen.run(generator, size)
```

**Arguments**

- **generator**: A generator
- **size**: The size parameter passed to the generation functions

**Details**

This is different to calling `generator$unGen(size)` in that it also works on (nested) lists of generators and pure values.

---

**gen.shrink**

*Helper to create a generator with a shrink function.*

**Description**

shrinker takes an `'a` and returns a vector of `'a`.

**Usage**

```
gen.shrink(shrinker, g)
```

**Arguments**

- **shrinker**: a function takes an `'a` and returning a vector of `'a`.
- **g**: a generator we wish to add shrinking to
gen.sized  

_Sized generator creation_

**Description**

Helper for making a gen with a size parameter. Pass a function which takes an int and returns a gen.

**Usage**

`gen.sized(f)`

**Arguments**

- `f`  
The function, taking a size and returning a generator

**Examples**

```
gen.sized ( function(e) gen.element(1:e) )
```

---

gen.structure  

_Generate a structure_

**Description**

If you can create an object with `structure`, you should be able to generate an object with this function from a generator or list of generators.

**Usage**

`gen.structure(x, ...)`

**Arguments**

- `x`  
  An object generator which will have various attributes attached to it.
- `...`  
  Attributes, specified in 'tag = value' form, which will be attached to generated data.

**Details**

`gen.structure` accepts the same forms of data as `forall`, and is flexible, in that any list of generators is considered to be a generator.
**Examples**

# To create a matrix
```r
gen.structure( gen.c(of = 6, gen.element(1:30)), dim = 3:2)
```

# To create a data frame for testing.
```r
gen.structure(
    list(
      gen.c(of = 4, gen.element(2:10)),
      gen.c(of = 4, gen.element(2:10)),
      c('a', 'b', 'c', 'd')
    ),
    names = c('a', 'b', 'constant'),
    class = 'data.frame',
    row.names = c('1', '2', '3', '4'))
```

---

**gen.unif**  
Generate a float between the from and to the values specified.

---

**Description**

Shrinks towards the from value, or if shrink.median is on, the middle.

**Usage**

```r
gen.unif(from, to, shrink.median = T)
```

**Arguments**

- `from` same as from in runif
- `to` same as to in runif
- `shrink.median` whether to shrink to the middle of the distribution instead of the low end.

**Examples**

```r
gen.unif(0, 1) # a float between 0 and 1
```

---

**generate**  
Compose generators

---

**Description**

Use ‘generator’ with a for loop over the output of another generator to create a new, more interesting generator.

**Usage**

```r
generate(loop)
```
Arguments

loop A ‘for’ loop expression, where the value iterated over is another Hedgehog generator.

See Also

[gen-monad()] for FP style ways of sequencing generators. This function is syntactic sugar over ‘gen.and_then’ to make it palatable for R users.

Examples

```r
gen_squares <- generate(for i in gen.int(10)) i^2
gen_sq_digits <- generate(for i in gen_squares) {
  gen.c(of = i, gen.element(1:9))
}
```

hedgehog Property based testing in R

Description

Hedgehog is a modern property based testing system in the spirit of QuickCheck, originally written in Haskell, but now also available in R.

Details

Software testing is critical when we want to distribute our work, but unit testing only covers examples we have thought of.

With hedgehog (integrated into testthat), we can instead test properties which our programs and functions should have, and allow automatic generation of tests, which cover more that we could imagine.

One of the key benefits of Hedgehog is integrated shrinking of counterexamples, which allows one to quickly find the cause of bugs, given salient examples when incorrect behaviour occurs.

Options

- ‘hedgehog.tests’: Number of tests to run in each property (Default: ‘100’).
- ‘hedgehog.size’: Maximum size parameter to pass to generators (Default: ‘50’).
- ‘hedgehog.shrinks’: Maximum number of shrinks to search for (Default: ‘100’).
- ‘hedgehog.discards’: Maximum number of discards permitted within a property test before failure (Default: ‘100’).

References


https://github.com/hedgehogqa/r-hedgehog
Examples

```r
library(hedgehog)
test_that("Reverse and concatenate symmetry", 
  forall(list(as = gen.c(gen.element(1:100))), 
    bs = gen.c(gen.element(1:100)))
  , function(as, bs)
    expect_identical(rev(c(as, bs)), c(rev(bs), rev(as)))
  )
)
```

**shrink.halves**

_Shrink a number by dividing it into halves._

**Description**

Shrink a number by dividing it into halves.

**Usage**

```r
shrink.halves(x)
```

**Arguments**

- `x` number to produce halves of

**Examples**

```r
shrink.towards(45)
# 22 11 5 2 1
```

**shrink.list**

_Shrink a list by edging towards the empty list._

**Description**

Shrink a list by edging towards the empty list.

**Usage**

```r
shrink.list(xs)
```

**Arguments**

- `xs` the list to shrink
shrink.removes  Produce permutations of removing num elements from a list.

Description

Produce permutations of removing num elements from a list.

Usage

shrink.removes(num, xs)

Arguments

num  the number of values to drop
xs   the list to shrink

shrink.towards  Shrink an integral number by edging towards a destination.

Description

Note we always try the destination first, as that is the optimal shrink.

Usage

shrink.towards(destination)

Arguments

destination  the value we want to shrink towards.

Examples

shrink.towards (0) (100)
# [0, 50, 75, 88, 94, 97, 99]

shrink.towards(500)(1000)
# [500, 750, 875, 938, 969, 985, 993, 997, 999]

shrink.towards (-50) (-26)
# [-50, -38, -32, -29, -27]
symbolic

| symbolic | A symbolic value. |

**Description**

These values are the outputs of a computation during the calculations’ construction, and allow a value to use the results of a previous function.

**Usage**

```plaintext
symbolic(var)
```

**Arguments**

- `var` the integer output indicator.

**Details**

Really, this is just an integer, which we use as a name for a value which will exist later in the computation.

---

**tree**

**Lazy rose trees**

**Description**

A rose tree is a type of multibranch tree. This is hedgehog’s internal implementation of a lazy rose tree.

**Usage**

```plaintext
tree(root, children_ = list())
tree.map(f, x)
tree.bind(f, x)
tree.liftA2(f, x, y)
tree.expand(shrink, x)
tree.unfold(shrink, a)
tree.unfoldForest(shrink, a)
tree.sequence(trees)
```
Arguments

- **root**: the root of the rose tree
- **children_**: a list of children for the tree.
- **f**: a function for mapping, binding, or applying
- **x**: a tree to map or bind over
- **y**: a tree to map or bind over
- **shrink**: a shrinking function
- **a**: a value to unfold from
- **trees**: a tree, or list or structure potentially containing trees to turn into a tree of said structure.

Details

In general, one should not be required to use any of the functions from this module as the combinators in the gen module should be expressive enough (if they’re not raise an issue).

---

**tree.replicate**  
*Creating trees of lists*

Description

Build a tree of a list, potentially keeping hold of an internal state.

Usage

```
tree.replicate(num, ma, ...)
tree.replicateS(num, ma, s, ...)
```

Arguments

- **num**: the length of the list in the tree
- **ma**: a function which (randomly) creates new tree to add to the list
- **...**: extra arguments to pass to the tree generating function
- **s**: a state used when replicating to keep track of.
<table>
<thead>
<tr>
<th>Command/Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>2</td>
</tr>
<tr>
<td>discard</td>
<td>3</td>
</tr>
<tr>
<td>expect_sequential</td>
<td>4</td>
</tr>
<tr>
<td>forall</td>
<td>4</td>
</tr>
<tr>
<td>gen (gen-monad)</td>
<td>7</td>
</tr>
<tr>
<td>gen-element</td>
<td>5</td>
</tr>
<tr>
<td>gen-monad</td>
<td>7</td>
</tr>
<tr>
<td>gen.actions</td>
<td>8</td>
</tr>
<tr>
<td>gen.and_then (gen-monad)</td>
<td>7</td>
</tr>
<tr>
<td>gen.beta</td>
<td>9</td>
</tr>
<tr>
<td>gen.bind (gen-monad)</td>
<td>7</td>
</tr>
<tr>
<td>gen.c</td>
<td>9</td>
</tr>
<tr>
<td>gen.choice (gen-element)</td>
<td>5</td>
</tr>
<tr>
<td>gen.date</td>
<td>10</td>
</tr>
<tr>
<td>gen.element (gen-element)</td>
<td>5</td>
</tr>
<tr>
<td>gen.example</td>
<td>10</td>
</tr>
<tr>
<td>gen.gamma</td>
<td>11</td>
</tr>
<tr>
<td>gen.impure</td>
<td>8</td>
</tr>
<tr>
<td>gen.impure (gen-monad)</td>
<td>7</td>
</tr>
<tr>
<td>gen.int (gen-element)</td>
<td>5</td>
</tr>
<tr>
<td>gen.list</td>
<td>11</td>
</tr>
<tr>
<td>gen.map (gen-monad)</td>
<td>7</td>
</tr>
<tr>
<td>gen.no.shrink</td>
<td>12</td>
</tr>
<tr>
<td>gen.pure (gen-monad)</td>
<td>7</td>
</tr>
<tr>
<td>gen.recursive</td>
<td>12</td>
</tr>
<tr>
<td>gen.run</td>
<td>13</td>
</tr>
<tr>
<td>gen.sample (gen-element)</td>
<td>5</td>
</tr>
<tr>
<td>gen.shrink</td>
<td>8, 13</td>
</tr>
<tr>
<td>gen.sized</td>
<td>13</td>
</tr>
<tr>
<td>gen.structure</td>
<td>14</td>
</tr>
<tr>
<td>gen.subsequence (gen-element)</td>
<td>5</td>
</tr>
<tr>
<td>gen.unif</td>
<td>15</td>
</tr>
<tr>
<td>gen.with (gen-monad)</td>
<td>7</td>
</tr>
<tr>
<td>generate</td>
<td>8, 15</td>
</tr>
<tr>
<td>hedgehog</td>
<td>16</td>
</tr>
<tr>
<td>hedgehog-package (hedgehog)</td>
<td>16</td>
</tr>
</tbody>
</table>

shrink.halves, 17
shrink.list, 17
shrink.removes, 18
shrink.towards, 18
symbolic, 19

tree, 19
tree.replicate, 20
tree.replicateS (tree.replicate), 20