Package: s7contract (via r-universe)

October 14, 2026

Title Behavioral Contracts and Generative Laws for 'S7'

Version 0.2.3.9000

Description Contract helpers built with 'S7' for expressing runtime protocols around ordinary 'S7' dispatch. Structural interfaces describe small sets of required 'S7' generics, while explicit traits record registered implementations with optional default methods and associated metadata. Optional runtime checks can validate argument and return specifications in contract-scoped evaluation. Generative laws combine generators, deterministic shrinking, and one-result 'tinytest' expectations.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 4.3.0)

Imports methods, S7, stats

Suggests knitr, rmarkdown, tinytest

VignetteBuilder knitr

URL https://github.com/sounkou-bioinfo/s7contract, https://sounkou-bioinfo.github.io/s7contract/

BugReports https://github.com/sounkou-bioinfo/s7contract/issues

Repository https://sounkou-bioinfo.r-universe.dev

Date/Publication 2026-09-14 20:17:20 UTC

RemoteUrl https://github.com/sounkou-bioinfo/s7contract

RemoteRef HEAD

RemoteSha 41dce50d482ad4bfac382e859662abcae0ba56c4

Contents

%::% . . . 2
gen_bind . . . 3
gen_commands . . . 5
gen_constant . . . 5
gen_double . . . 6
gen_element . . . 7
gen_example . . . 8
gen_sample . . . 9
interface_requirements . . . 10
new_command . . . 11
new_generator . . . 12
new_interface . . . 13
new_law . . . 14
new_state_law . . . 17
new_trait . . . 18
s7contract . . . 20
trait_methods . . . 21

Index **23**

`%::%` *Evaluate an S7 call under an interface or trait contract*

Description

`with(contract, expr)` and `expr %::% contract` evaluate `expr` in a contract mask. Required generics are shadowed by checking wrappers, so calls to those generics use normal S7 dispatch while checking the optional argument and return specifications stored in an interface requirement or trait method. Only names resolved through the mask are checked; namespace-qualified calls and calls inside separately defined helpers are not instrumented.

Usage

```
expr %::% contract
```

Arguments

`expr`	An expression evaluated in a contract mask. Calls to generics named in the contract are checked.
`contract`	An interface created by `new_interface()` or a trait created by `new_trait()`.

Details

Checks force dispatch and typed arguments before the generic body runs. Generic defaults retain their lexical scope and share ordinary R promises. If a typed argument has only a method default, that default is evaluated before dispatch and supplied to the generic. Such defaults should be pure expressions of arguments and lexical bindings, without relying on method-body locals or `missing()` for that argument.

Value

The value of `expr`, after any optional return check.

Examples

```
local({
  draw <- S7::new_generic("draw", "x", function(x, color) {
    S7::S7_dispatch()
  })
  Circle <- S7::new_class("TypedCircle", properties = list(r = S7::class_double))
  S7::method(draw, Circle) <- function(x, color) paste(color, x@r)
  Drawable <- new_interface(
    "TypedDrawable",
    generics = list(draw = interface_requirement(
      draw,
      args = list(color = S7::class_character),
      returns = S7::class_character
    ))
  )
  with(Drawable, draw(Circle(r = 2), color = "red"))
  checked_draw <- with(Drawable, function(x) draw(x, color = "red"))
  checked_draw(Circle(r = 2))
  draw(Circle(r = 2), color = "red") %::% Drawable
})
```

---

gen_bind *Compose dependent, sized, and recursive generators*

Description

`gen_bind()` passes a generated value to `bind`, which constructs the next generator. Shrinking first rebuilds that generator for smaller source values, then shrinks its output. Each rebuild uses the same locally captured seed and size, so random downstream draws remain reproducible regardless of shrink traversal or random draws made by a law. The surrounding RNG state is restored after each rebuild. Callbacks must not depend on external mutable state or change the RNG configuration.

Usage

```
gen_bind(generator, bind, prototype = list())
```

```
gen_sized(factory, prototype = list())

gen_resize(generator, size)

gen_recursive(base, expand, prototype = list())
```

Arguments

generator, base	A generator. base produces non-recursive values.
bind	Function of one generated value returning a generator.
prototype	Zero-length prototype of generated values, used when this generator is an element of gen_vector(). Defaults to list elements.
factory	Function of one non-negative integer size returning a generator.
size	Non-negative integer size to use for every draw.
expand	Function accepting a child generator and returning a generator for one recursive layer.

Details

gen_sized() passes the current size to a generator factory. gen_resize() fixes the size used by one generator without changing its siblings.

gen_recursive() chooses between base and the generator returned by expand(child). The supplied child recursively uses half the current size, rounded down; size zero draws only from base. Shrinking tries the base branch before shrinking the expanded value. Recursion through child therefore terminates, provided callbacks themselves terminate and do not introduce other recursion. Size bounds recursion depth, not total node count.

Value

An S7 generator object.

Examples

```
sized_vectors <- gen_bind(gen_integer(1L, 8L), function(n) {
  gen_product(n = gen_constant(n), x = gen_vector(gen_integer(), n, n))
})
gen_example(sized_vectors)

trees <- gen_recursive(gen_constant(0L), function(child) {
  gen_product(left = child, right = child)
})
gen_example(trees, size = 4L)
```

gen_commands *Generate and shrink model-valid command sequences*

Description

Sequence length is sampled from zero through `min(size, max)`. Generation stops early if every command is unavailable. Available commands are selected with equal probability. Generated inputs must satisfy their preconditions.

Usage

```
gen_commands(initial, commands, max = 10L)
```

Arguments

`initial`	Initial model with value semantics.
`commands`	Non-empty list of descriptors made with `new_command()`.
`max`	Maximum sequence length.

Details

Shrinking removes contiguous chunks, then shrinks command inputs. Each candidate is replayed against the symbolic model: commands with unsatisfied preconditions or missing output references are removed, together with any commands that depend on them. Retained command IDs never change. Candidate construction is lazy and executes no implementation commands or fixtures.

Value

An S7 generator of lists of steps. Each step contains an integer `id`, a `command` name, and its generated `input`.

gen_constant *Basic property-based test generators*

Description

These generators carry their own deterministic shrink trees. Integer ranges and vector lengths expand with the runner's size. Integer values shrink toward zero when zero is within bounds, or toward the nearest bound. Product generators shrink one component at a time in argument order. Vector generators return an atomic vector only when the element prototype is atomic; those element draws must be scalar and match the prototype's storage type. Otherwise, vector generators return a list with one entry per element draw. Nested vector generators therefore return lists of vectors. Vector shrinking removes contiguous chunks, then shrinks individual elements, preserving the minimum length.

Usage

```
gen_constant(value)

gen_integer(min = -100L, max = 100L)

gen_map(generator, transform, prototype = list())

gen_product(...)

gen_vector(element, min = 0L, max = 10L)
```

Arguments

`value`	Constant value to generate.
`min`, `max`	Inclusive integer bounds. For `gen_vector()`, bounds on vector length.
`generator`, `element`	A generator.
`transform`	Function applied to generated values.
`prototype`	Zero-length prototype of mapped values.
`...`	Uniquely named generators.

Value

An S7 generator object.

Examples

```
pairs <- gen_product(x = gen_integer(), y = gen_integer())
vectors <- gen_vector(gen_integer(), min = 0L, max = 8L)
```

---

gen_double *Generate finite double values*

Description

At size zero, `gen_double()` draws only `origin`. The bounds expand linearly from the origin to `min` and `max`, reaching the full interval at size 100. Larger sizes use that same interval. Use `gen_resize()` to sample the full range at every runner size. Within the current bounds, generation interpolates one uniform draw from `stats::runif()`. This samples a finite-precision approximation to a continuous uniform distribution, not all representable doubles uniformly. Bounds are inclusive constraints; endpoints are not guaranteed to be drawn. Use `gen_choice()` with constants to target them.

Usage

```
gen_double(min = -100, max = 100, origin = NULL)
```

Arguments

`min`, `max`	Finite scalar numeric bounds, with `min` <= `max`.
`origin`	Finite scalar numeric shrink target within the bounds. `NULL` chooses zero when it is in range, otherwise the nearest bound.

Details

Shrinking tries the origin, then the midpoint between the origin and the generated value, then successive midpoints approaching that value. For example, 8 with origin 0 has children 0, 4, 6, 7, 7.5, and so on. Each child follows the same rule. Candidates remain between the origin and their parent; iteration stops when rounding prevents further progress. Children are built only when visited. Values close to zero can require many steps to shrink through subnormal doubles, so the runner's evaluation budget still applies.

`NA`, `NaN`, and infinities are excluded. Add them explicitly with `gen_choice()` or `gen_element()`. Branch weights control generation and branch order controls shrinking; zero-weight branches are excluded from both.

Value

An S7 generator with a double element prototype.

References

Haskell Hedgehog separates shrink origins from size-dependent bounds and uses fractional shrinking toward an origin. The R Hedgehog manual documents `gen.unif()` and mixtures with exceptional numeric values.

Examples

```
measurements <- gen_double(-10, 10)
gen_example(measurements, size = 100L)

# 80% finite draws; 5% each for NA, NaN, -Inf, and Inf.
numeric_values <- gen_choice(
  measurements, gen_element(c(NA_real_, NaN, -Inf, Inf)),
  prob = c(4, 1)
)
gen_example(gen_vector(numeric_values, max = 5L))
```

---

gen_element *Choose values or generators*

Description

`gen_element()` samples one element of a vector or list. `gen_choice()` samples a generator and draws from it at the current size. All choices are available at size zero. Both shrink toward earlier entries, then `gen_choice()` shrinks within the selected generator. Zero-weight entries are excluded from both generation and shrinking.

Usage

```
gen_element(values, prob = NULL)

gen_choice(..., prob = NULL)
```

Arguments

values	Non-empty atomic vector or list of values.
prob	Optional finite non-negative sampling weights, one per entry, with at least one positive weight.
...	One or more generators, ordered from simpler to more complex.

Value

An S7 generator object. `gen_element()` preserves an atomic input's element prototype; list inputs use a list prototype. `gen_choice()` retains a common prototype when all branches agree, otherwise it uses list elements.

Examples

```
bases <- gen_element(c("A", "C", "G", "T"))
nullable <- gen_choice(gen_constant(NA_integer_), gen_integer(), prob = c(1, 9))
gen_example(gen_vector(bases, min = 4L, max = 4L))
```

---

gen_example *Inspect a generator or disable its shrinking*

Description

`gen_example()` draws one value at a fixed size with a local seed, restoring the caller's RNG kind and state on exit, as `check_law()` does. It does not expand the shrink tree. `gen_no_shrink()` keeps generation unchanged but removes all shrink candidates, including those carried by composed generators.

Usage

```
gen_example(generator, size = 10L, seed = 1L)

gen_no_shrink(generator)
```

Arguments

generator	A generator.
size	Non-negative integer size.
seed	Integer seed for reproducible generation.

Value

`gen_example()` returns one generated value. `gen_no_shrink()` returns an S7 generator object with the original element prototype.

Examples

```
gen_example(gen_vector(gen_integer()), size = 5L, seed = 42L)
```

`gen_sample` *Sample source positions without replacement*

Description

`gen_sample()` draws a uniform ordered sample of exactly `size` source positions, without replacement. Its cardinality and sampling range do not depend on the runner's size. The default draws permutations of the source. Shrinking moves positions toward the beginning of the source, from left to right. Each position tries the earliest position unused by its prefix, then integer bisections toward its current position. Targets already in the prefix are skipped; targets used later in the sample are swapped. Every child is lexicographically smaller, with the same cardinality and distinct positions. The terminal sample consists of the first `size` source entries in order.

Usage

```
gen_sample(values, size = length(values))

gen_subsequence(values, min = 0L, max = length(values))
```

Arguments

`values`	An atomic vector or list, optionally named. `NULL` is also accepted as an empty source. Classed vectors such as dates and factors must support `length()` and integer `[` subsetting that preserves their class and returns one entry per position. Arrays and data frames are not supported. Length must be at most `.Machine$integer.max`.
`size`	Fixed non-negative sample cardinality, at most `length(values)`.
`min`, `max`	Inclusive non-negative subsequence length bounds, with `min <= max <= length(values)`.

Details

`gen_subsequence()` chooses a length uniformly between `min` and `min(max, min + runner_size)`, then samples that many positions and sorts them. It shrinks length toward `min` first, rebuilding a sample with a captured seed as in `gen_bind()`, then shrinks positions. Source order and length bounds are preserved. Different position shrinks can yield the same sorted subsequence. Setting `min = max = length(values)` yields a constant.

Uniqueness concerns positions, not values: duplicated source entries can appear together. Subsetting with `[` preserves names and supported classes; values themselves are not shrunk. Empty sources allow only empty selections. Both generators have a list prototype, so `gen_vector()` nests their results. For sampling with replacement, compose `gen_element()` and `gen_vector()`.

Value

An S7 generator.

References

R's sampling documentation describes positional sampling and the hash algorithm used for small samples from large populations. These generators use `sample.int()` without constructing a vector of every source position. The R Hedgehog manual documents subsequences and sampling as distinct generator domains.

Examples

```
gen_example(gen_sample(letters, size = 3L))
gen_example(gen_subsequence(letters, max = 5L))
gen_example(gen_sample(seq_len(100000000L), size = 3L))
```

`interface_requirements`

Inspect or check a Go-like structural interface

Description

Inspect or check a Go-like structural interface

Usage

```
interface_requirements(interface, inherited = TRUE)

interface_report(x, interface)

missing_requirements(x, interface)

implements(x, interface)

assert_implements(x, interface, arg = deparse(substitute(x)))

as_interface(x, interface)
```

Arguments

`interface`	An interface created by `new_interface()`.
`inherited`	Include inherited requirements from parent interfaces?
`x`	An S7 object or class, S3 object or class wrapper, S4 object or class, or a value or wrapper for a supported S7 base class.
`arg`	Name to use in error messages.

Value

`interface_requirements()` returns a named list of `interface_requirement()` objects. `interface_report()` and `missing_requirements()` return data frames. `implements()` returns a single logical value. `assert_implements()` and `as_interface()` return `x`, unchanged.

`new_command` *Describe a command for a stateful protocol*

Description

Commands separate the implementation's effects from a reference model. `generate(state)` returns an input generator, or `NULL` when the command is unavailable. `execute(fixture, input)` calls the implementation. `require(state, input)` checks the symbolic precondition, and `ensure(state, input, output)` checks the observed result against the model before the command. Both predicates return one non-missing logical value.

Usage

```
new_command(
  name,
  generate,
  execute,
  ensure,
  update = function(state, input, output) state,
  require = function(state, input) TRUE
)
```

Arguments

`name`	Non-empty command name, unique within a command set.
`generate`	Function of the model state returning a generator or `NULL`.
`execute`	Function of the fixture and resolved input returning an output.
`ensure`	Function of the previous state, resolved input, and output returning whether the postcondition holds.
`update`	Function of the previous state, input, and output returning the next state. Defaults to leaving the model unchanged.
`require`	Function of the symbolic state and input returning whether the command is permitted. Defaults to `TRUE`.

Details

`update(state, input, output)` returns the next model. During generation and shrinking, `output` is an opaque reference to this command's future result. During execution it is the actual result. Updates may store and pass outputs but must not inspect or compute with them. Expected values should come from the model and inputs, independently of the implementation.

References may be passed as inputs directly or inside ordinary, unclassed lists. They are resolved before execution, including references to `NULL` outputs. References embedded in other objects are not traversed. Model values must have value semantics: callbacks must not mutate shared environments or other reference objects in the model. Generation, preconditions, and updates must be pure apart from generator draws; preconditions and updates must not draw random numbers.

Value

An S7 command descriptor, used by `gen_commands()` and `new_state_law()`.

`new_generator` *Construct a property-based test generator*

Description

A generator draws a value together with an integrated tree of smaller values. `new_generator()` is the extension point for custom generators. Its `draw` function receives a non-negative integer size and returns one value. Its deterministic `shrink` function returns a list of strictly smaller values. The custom shrinker constructs that list itself; the framework constructs and transforms the corresponding tree nodes only as they are visited.

Usage

```
new_generator(
  draw,
  shrink = function(value) list(),
  label = "custom",
  prototype = list()
)
```

Arguments

`draw`	Function of one `size` argument that returns a value.
`shrink`	Function of one generated value that returns a list of smaller values.
`label`	Short description used in diagnostics.
`prototype`	Zero-length prototype used by `gen_vector()`.

Value

An S7 generator object.

new_interface *Build a Go-like structural interface on top of S7*

Description

new_interface() models the method-list part of Go interfaces as a list of required S7 generics. An interface is just a named set of required generics, and a class or object satisfies it when S7 can find a method for every required generic.

Usage

```
new_interface(
  name,
  generics = list(),
  parents = list(),
  package = NULL,
  methods = NULL
)

interface_requirement(
  generic,
  name = NULL,
  args = list(),
  returns = S7::class_any
)
```

Arguments

name	For new_interface(), the interface name. For interface_requirement(), the requirement name; it defaults to the generic name when omitted.
generics	For new_interface(), a named list of S7 generics or interface_requirement() objects. These are generic functions because S7 methods are registered separately on generics.
parents	Optional interface or list of interfaces to embed.
package	Optional package name used only for display.
methods	Compatibility alias for generics.
generic	An S7 generic function.
args	Optional named list of S7 classes, interfaces, or traits for runtime argument checking with with() or %::%. Arguments named in args are also checked against generic and method formals during conformance checks. Dispatch arguments other than the first can use S7 classes or unions to refine multiple-dispatch requirements. Every concrete combination in those unions must have a compatible method.
returns	Optional S7 class, interface, or trait for runtime return checking with with() or %::%; defaults to S7::class_any.

Details

Embedding retains every parent requirement. Requirements sharing an alias or a generic name must have identical generics and type specifications; incompatible collisions are errors. Both aliases and generic names refer to the checked operation inside `with()`. Reusing a requirement across parents is valid, including diamond-shaped inheritance.

This mirrors Go's basic interfaces defined only by methods. Define small interfaces at the point where consuming code needs a behavior. Define S7 classes, generics, and methods normally; then let consumers name the protocol they accept. Up-front interfaces are also useful for package protocols, abstract data types, or recursive protocols.

Go's full post-1.18 type-set language is outside this model, including tilde type terms, unions of concrete types, or pointer/value receiver rules.

Value

`new_interface()` returns an S7 object of class `s7_interface`. `interface_requirement()` returns an S7 object of class `s7_interface_requirement`.

Examples

```
local({
  area <- S7::new_generic("area", "x")
  draw <- S7::new_generic("draw", "x")

  Circle <- S7::new_class(
    "Circle",
    properties = list(r = S7::class_double)
  )
  Rect <- S7::new_class(
    "Rect",
    properties = list(w = S7::class_double, h = S7::class_double)
  )

  S7::method(area, Circle) <- function(x) pi * x@r^2
  S7::method(draw, Circle) <- function(x) sprintf("circle(r = %s)", x@r)
  S7::method(area, Rect) <- function(x) x@w * x@h

  Drawable <- new_interface("Drawable", generics = list(draw = draw))
  Shape <- new_interface("Shape", generics = list(area = area), parents = Drawable)

  implements(Circle, Shape)
  missing_requirements(Rect, Shape)
})
```

`new_law` *Define and check a generative law*

Description

A law is a named, universally quantified claim sampled over explicitly supplied generators. `check_law()` is test-framework neutral and returns a structured S7 result. `expect_law()` adapts that result to one tinytest expectation, regardless of how many generated cases were checked.

Usage

```
new_law(
  name,
  generators,
  holds,
  classify = function(...) character(),
  min_coverage = numeric()
)

assume(condition)

check_law(
  law,
  tests = getOption("s7contract.tests", 100L),
  seed = getOption("s7contract.seed", 1L),
  shrinks = getOption("s7contract.shrinks", 100L),
  discards = getOption("s7contract.discards", 100L),
  max_size = getOption("s7contract.max_size", 100L)
)

format_check_result(x)

expect_law(
  law,
  tests = getOption("s7contract.tests", 100L),
  seed = getOption("s7contract.seed", 1L),
  shrinks = getOption("s7contract.shrinks", 100L),
  discards = getOption("s7contract.discards", 100L),
  max_size = getOption("s7contract.max_size", 100L)
)
```

Arguments

`name`	Non-empty description of the law.
`generators`	Uniquely named non-empty list of generators.
`holds`	Function accepting the generated arguments and returning one non-missing logical value.
`classify`	Function accepting the same generated arguments as `holds` and returning character labels, or `NULL` for none. Duplicate labels count once per case; names are ignored. Labels must be non-missing and non-empty. The classifier must be deterministic, must not draw random numbers, and must not mutate inputs or external state. It is never called on shrinks.

min_coverage	Named numeric vector of minimum proportions in [0, 1], with unique label names. For example, c(nonempty = 0.5) requires at least half of accepted generated cases to carry the label "nonempty".
condition	Scalar logical precondition.
law	A law created by new_law().
tests	Number of passing cases required.
seed	Deterministic local random seed. The caller's RNG kind and state are restored after the run.
shrinks	Maximum number of candidate shrink evaluations.
discards	Maximum number of discarded generated cases.
max_size	Maximum size passed to generators.
x	A result returned by check_law().

Details

Runs use Mersenne-Twister, Inversion normals, and Rejection sampling, independently of the caller's RNG kind. Box-Muller callers are rejected before any RNG state is changed because R does not expose their cached normal draw. Replay requires unchanged generator/law code, run parameters, and compatible R/package versions; generators and laws must not depend on external mutable state or change the RNG configuration. The result's parameters list records all run arguments except law, for use with do.call(check_law, ...).

Shrinking is an ordered search, not a guarantee of a global minimum. The counterexample's minimal field holds the last accepted failing candidate on that search path; candidates have no general size ordering. A result's shrink_status is "complete" when no immediate child preserves the failure, "budget" when the evaluation limit stopped the search (including zero), "error" if constructing candidates failed, or "not_needed" when no counterexample was found. A shrinking error or warning is stored separately in shrink_condition; the original and last failing examples are retained. Generator warnings and errors terminate the run with status "error". Warnings or errors from holds stop that evaluation and are counterexamples. A subsequent discard cannot hide a warning. Stateful laws created by new_state_law() additionally retain failure traces in the counterexample's original_condition and condition fields. Callback defects stop their shrink search, preserving any earlier false postcondition.

Optional classify labels each generated input before holds runs. Labels count once per case whose outcome is a pass or a false postcondition, including stateful postcondition failures. Discards, errors, and shrink candidates are excluded. A classifier warning, error, or invalid return terminates the run with status "error" before evaluating that case's law.

The result's coverage data frame contains label, count, proportion, minimum, and met; coverage_cases is the denominator. Requirements for unseen labels have count zero. Unrequested minima and their met values are NA; with no accepted cases, proportions and all met values are also NA. After the requested passing cases, unmet minima give status "insufficient_coverage" and make expect_law() fail without a counterexample. Falsification, error, and exhaustion retain their own statuses and report partial coverage. Minima describe observed proportions within the test budget, without a statistical confidence guarantee. Generator size and preconditions can change the sampled distribution.

In a tinytest file, call tinytest::using(s7contract) before calling expect_law(). This activates tinytest's supported extension capture so the property run is recorded as one ordinary test result.

Value

`new_law()` returns an S7 law. `check_law()` returns an S7 check result. `expect_law()` returns one `tinytest` result. `assume()` returns invisibly when its condition is true and otherwise discards the case.

Examples

```
reverse_law <- new_law(
  "reverse is involutive",
  generators = list(x = gen_vector(gen_integer(), max = 8L)),
  holds = function(x) identical(rev(rev(x)), x)
)
check_law(reverse_law, tests = 20L, seed = 1L)
```

---

new_state_law *Define a generative law for a stateful protocol*

Description

Builds an ordinary law for `check_law()` or `expect_law()`. Each evaluation, including every shrink candidate, calls `setup()` to obtain a fresh fixture and `teardown(fixture)` once after successful setup. If setup itself fails, it is responsible for releasing any partially acquired resources.

Usage

```
new_state_law(
  name,
  initial,
  commands,
  setup,
  teardown = function(fixture) NULL,
  max_commands = 10L,
  classify = function(...) character(),
  min_coverage = numeric()
)
```

Arguments

`name`	Non-empty description of the law.
`initial`	Initial model with value semantics.
`commands`	Non-empty list of descriptors made with `new_command()`.
`setup`	Function of no arguments returning a fresh implementation fixture.
`teardown`	Function of that fixture releasing its resources.
`max_commands`	Maximum generated sequence length.
`classify`	Function of the generated `sequence` returning case labels, as in `new_law()`. Labels describe the generated sequence, including any suffix not executed after a failure.
`min_coverage`	Named minimum case proportions, as in `new_law()`.

Details

A false command postcondition falsifies the law. Unexpected callback errors or warnings produce an error and stop shrinking. Cleanup failures do not replace an established postcondition failure: they are recorded in its `cleanup_condition` and stop shrinking. User callbacks must terminate; the shrink budget counts candidate evaluations, not individual commands.

Counterexamples retain the original and reduced command sequences. Their `original_condition` and `condition` describe the original and reduced failures, with the failing `step`, `command`, callback `phase`, pre-command `model`, resolved `input`, `output`, and execution `trace`. Trace entries record the model before and after each completed command. Model and output snapshots require value semantics; mutable output handles retain their usual R reference semantics. Replay has the same requirements as `check_law()`, and setup must reproduce the same initial implementation state.

Value

An S7 law accepted by `check_law()` and `expect_law()`.

Examples

```
increment <- new_command(
  "increment",
  generate = function(state) gen_integer(0L, 5L),
  execute = function(fixture, input) {
    fixture$value <- fixture$value + input
    fixture$value
  },
  update = function(state, input, output) state + input,
  ensure = function(state, input, output) identical(output, state + input)
)
counter_law <- new_state_law(
  "counter follows its model", 0L, list(increment),
  setup = function() list2env(list(value = 0L), parent = emptyenv())
)
check_law(counter_law, tests = 20L, seed = 1L)
```

`new_trait` *Build a Rust-like explicit trait on top of S7*

Description

`new_trait()` adds a nominal contract registry on top of S7 dispatch. A class only has the trait after `impl_trait()` records the implementation, even if compatible S7 methods already exist. Registrations belong to the current R session and the particular trait descriptor, not its display name. Reuse the same descriptor for registration and checks; deserializing a descriptor does not transfer its registrations. S7 subclasses and S3 subclasses need their own registration; method inheritance alone does not confer a trait.

Usage

```
new_trait(
  name,
  methods = list(),
  parents = list(),
  assoc_types = character(),
  assoc_consts = list(),
  package = NULL
)

trait_method(
  generic,
  default = NULL,
  name = NULL,
  args = list(),
  returns = S7::class_any
)
```

Arguments

name	For `new_trait()`, the trait name. For `trait_method()`, the method name; it defaults to the generic name when omitted.
methods	For `new_trait()`, a named list of S7 generics or `trait_method()` objects.
parents	Optional trait or list of supertraits.
assoc_types	Required associated type names, or a named list of default associated type values.
assoc_consts	Required associated constant names, or a named list of default constant values.
package	Optional package name used only for display.
generic	An S7 generic function.
default	Optional default implementation. If supplied, `impl_trait()` uses it when a class does not provide an override for that method.
args	Optional named list of S7 classes, interfaces, or traits for runtime argument checking with `with()` or `%::%`. Dispatch arguments other than the first can use S7 classes or unions to refine multiple-dispatch requirements.
returns	Optional S7 class, interface, or trait for runtime return checking with `with()` or `%::%`; defaults to `S7::class_any`.

Details

Method aliases and generic names share the scoped call namespace. Overlapping requirements must have identical generics, defaults and type specifications. Associated items must have one declaring trait: a diamond can share the same declaration, but independent supertraits cannot declare the same item name within `assoc_types` or within `assoc_consts`. Inherited values are read from the declaring supertrait's implementation.

This makes default methods and associated metadata practical, but the result remains a runtime R abstraction. It does not emulate Rust's compile-time trait bounds, coherence, orphan rules, or type-checked associated types.

Value

`new_trait()` returns an S7 object of class `s7_trait`. `trait_method()` returns an S7 object of class `s7_trait_method`.

Examples

```
local({
  area <- S7::new_generic("area", "x")
  perimeter <- S7::new_generic("perimeter", "x")

  Circle <- S7::new_class(
    "Circle",
    properties = list(r = S7::class_double)
  )

  Measurable <- new_trait(
    "Measurable",
    methods = list(
      area = trait_method(area),
      perimeter = trait_method(perimeter, default = function(x) NA_real_)
    ),
    assoc_consts = c("UNITS")
  )

  impl_trait(
    Measurable,
    Circle,
    methods = list(area = function(x) pi * x@r^2),
    assoc_consts = list(UNITS = "unitless")
  )

  has_trait(Circle, Measurable)
  trait_call(Measurable, "area", Circle(r = 2))
  trait_assoc_const(Measurable, Circle, "UNITS")
})
```

`s7contract` *s7contract: Behavioral Contracts and Generative Laws for S7*

Description

`s7contract` makes behavioral protocols explicit and testable around ordinary S7 dispatch:

Details

- Go-like structural interfaces defined by required generics.
- Rust-like explicit traits with default methods and associated metadata.
- Optional argument and return specifications checked at the point of use.

- Property-based laws with integrated shrinking and tinytest expectations.

`implements()` checks method availability and `has_trait()` checks declared implementation. `check_law()` tests behavior over generated cases. Protocol authors can reuse laws across implementations by writing functions that construct lists of laws; see `vignette("protocol-laws")`. `new_state_law()` tests sequences of commands against a reference model with fresh fixtures.

Author(s)

Maintainer: Sounkou Mahamane Toure <sounkoutoure@gmail.com>

See Also

Useful links:

- https://github.com/sounkou-bioinfo/s7contract
- https://sounkou-bioinfo.github.io/s7contract/
- Report bugs at https://github.com/sounkou-bioinfo/s7contract/issues

---

`trait_methods` *Inspect or use a Rust-like explicit trait*

Description

Methods are validated by S7 on isolated tables before registration is published. Method bindings and the implementation record are published together; a publication error restores the touched bindings. Aliases for the same generic must supply identical implementation functions.

Usage

```
trait_methods(trait, inherited = TRUE)

impl_trait(
  trait,
  class,
  methods = list(),
  assoc_types = list(),
  assoc_consts = list(),
  replace = FALSE
)

trait_report(x, trait)

has_trait(x, trait)

assert_trait(x, trait, arg = deparse(substitute(x)))

trait_call(trait, method, x, ...)
```

```
trait_assoc_type(trait, x, name)

trait_assoc_const(trait, x, name)
```

Arguments

trait	A trait created by new_trait().
inherited	Include inherited methods from supertraits?
class	A concrete S7 class, S3 class wrapper, S4 class, or base class wrapper. Register union members separately. Dispatch wildcards (class_any and class_missing) are not concrete targets.
methods	Named list of method implementations. Omitted trait methods use their default implementation when one is available.
assoc_types	Named list of associated type values.
assoc_consts	Named list of associated constant values.
replace	Replace an existing implementation record and silence warnings about visible S7 methods?
x	An S7, S3, S4 or supported base-class object, or its class descriptor.
arg	Name to use in error messages.
method	Method name within the trait.
...	Additional arguments passed to the S7 generic.
name	Associated item name.

Value

trait_methods() returns a named list of trait_method() objects. impl_trait() returns the stored implementation record, invisibly. trait_report() returns a one-row data frame. has_trait() returns a single logical value. assert_trait() returns x, unchanged. trait_call() returns the result of the underlying S7 generic. trait_assoc_type() and trait_assoc_const() return the stored associated item value.

Index

%::%, 2

as_interface *(*interface_requirements*)*, 10
assert_implements *(*interface_requirements*)*, 10
assert_trait *(*trait_methods*)*, 21
assume *(*new_law*)*, 14

check_law *(*new_law*)*, 14
check_law(), *8*, *17*, *18*, *21*
contract_syntax *(*%::%*)*, 2

expect_law *(*new_law*)*, 14
expect_law(), *17*, *18*

format_check_result *(*new_law*)*, 14

gen_bind, 3
gen_bind(), *9*
gen_choice *(*gen_element*)*, 7
gen_choice(), *6*, *7*
gen_commands, 5
gen_commands(), *12*
gen_constant, 5
gen_double, 6
gen_element, 7
gen_element(), *7*, *9*
gen_example, 8
gen_integer *(*gen_constant*)*, 5
gen_map *(*gen_constant*)*, 5
gen_no_shrink *(*gen_example*)*, 8
gen_product *(*gen_constant*)*, 5
gen_recursive *(*gen_bind*)*, 3
gen_resize *(*gen_bind*)*, 3
gen_resize(), *6*
gen_sample, 9
gen_sized *(*gen_bind*)*, 3
gen_subsequence *(*gen_sample*)*, 9
gen_vector *(*gen_constant*)*, 5
gen_vector(), *4*, *9*, *12*

has_trait *(*trait_methods*)*, 21
has_trait(), *21*

impl_trait *(*trait_methods*)*, 21
implements *(*interface_requirements*)*, 10
implements(), *21*
interface_report *(*interface_requirements*)*, 10
interface_requirement *(*new_interface*)*, 13
interface_requirements, 10

missing_requirements *(*interface_requirements*)*, 10

new_command, 11
new_command(), *5*, *17*
new_generator, 12
new_interface, 13
new_interface(), *2*
new_law, 14
new_law(), *17*
new_state_law, 17
new_state_law(), *12*, *16*, *21*
new_trait, 18
new_trait(), *2*

s7contract, 20
s7contract-package *(*s7contract*)*, 20
stats::runif(), *6*

trait_assoc_const *(*trait_methods*)*, 21
trait_assoc_type *(*trait_methods*)*, 21
trait_call *(*trait_methods*)*, 21
trait_method *(*new_trait*)*, 18
trait_methods, 21
trait_report *(*trait_methods*)*, 21

with(), *14*