Package 'paradox'

July 23, 2025

Type Package

Title Define and Work with Parameter Spaces for Complex Algorithms

Version 1.0.1

Description Define parameter spaces, constraints and dependencies for arbitrary algorithms, to program on such spaces. Also includes statistical designs and random samplers. Objects are implemented as 'R6' classes.

License LGPL-3

URL https://paradox.mlr-org.com, https://github.com/mlr-org/paradox

BugReports https://github.com/mlr-org/paradox/issues

Imports backports, checkmate, data.table, methods, mlr3misc (>= 0.9.4), R6

Suggests rmarkdown, mlr3learners, e1071, knitr, lhs, spacefillr, testthat

Encoding UTF-8

Config/testthat/edition 3

Config/testthat/parallel false

NeedsCompilation no

RoxygenNote 7.3.2

VignetteBuilder knitr

Collate 'Condition.R' 'Design.R' 'Domain.R' 'Domain_methods.R' 'NoDefault.R' 'ParamDbl.R' 'ParamFct.R' 'ParamInt.R' 'ParamLgl.R' 'ParamSet.R' 'ParamSetCollection.R' 'ParamUty.R' 'Sampler.R' 'Sampler1D.R' 'SamplerHierarchical.R' 'SamplerJointIndep.R' 'SamplerUnif.R' 'asserts.R' 'default_values.R' 'generate_design_grid.R' 'generate_design_lhs.R' 'generate_design_random.R' 'generate_design_sobol.R' 'helper.R' 'ps.R' 'ps_replicate.R' 'ps_union.R' 'reexports.R' 'to_tune.R' 'zzz.R'

Author Michel Lang [aut] (ORCID: <https://orcid.org/0000-0001-9754-0393>),
Bernd Bischl [aut] (ORCID: <https://orcid.org/0000-0001-6002-6980>),
Jakob Richter [aut] (ORCID: <https://orcid.org/0000-0003-4481-5554>),
Xudong Sun [aut] (ORCID: <https://orcid.org/0000-0003-3269-2307>),
Martin Binder [aut, cre],
Marc Becker [ctb] (ORCID: <https://orcid.org/0000-0002-8115-0400>)

Maintainer Martin Binder <mlr.developer@mb706.com>

Repository CRAN

Date/Publication 2024-07-09 13:10:02 UTC

Contents

paradox-package . . . 3
assert_param_set . . . 3
condition_test . . . 4
default_values . . . 5
Design . . . 6
Domain . . . 7
generate_design_grid . . . 13
generate_design_lhs . . . 14
generate_design_random . . . 15
generate_design_sobol . . . 15
NO_DEF . . . 16
ParamSet . . . 17
ParamSetCollection . . . 28
ps . . . 30
psc . . . 32
ps_replicate . . . 32
ps_union . . . 34
Sampler . . . 35
Sampler1D . . . 37
Sampler1DCateg . . . 38
Sampler1DNormal . . . 39
Sampler1DRfun . . . 40
Sampler1DUnif . . . 41
SamplerHierarchical . . . 42
SamplerJointIndep . . . 43
SamplerUnif . . . 44
to_tune . . . 45

Index **49**

paradox-package *paradox: Define and Work with Parameter Spaces for Complex Algorithms*

Description

Define parameter spaces, constraints and dependencies for arbitrary algorithms, to program on such spaces. Also includes statistical designs and random samplers. Objects are implemented as 'R6' classes.

Author(s)

Maintainer: Martin Binder <mlr.developer@mb706.com>

Authors:

- Michel Lang <michellang@gmail.com> (ORCID)
- Bernd Bischl <bernd_bischl@gmx.net> (ORCID)
- Jakob Richter <jakob1richter@gmail.com> (ORCID)
- Xudong Sun <smilesun.east@gmail.com> (ORCID)

Other contributors:

- Marc Becker <marcbecker@posteo.de> (ORCID) [contributor]

See Also

Useful links:

- https://paradox.mlr-org.com
- https://github.com/mlr-org/paradox
- Report bugs at https://github.com/mlr-org/paradox/issues

assert_param_set *Assertions for Params and ParamSets*

Description

Assertions for Params and ParamSets

Usage

```
assert_param_set(
  param_set,
  cl = NULL,
  no_untyped = FALSE,
  must_bounded = FALSE,
  no_deps = FALSE
)
```

Arguments

param_set	(ParamSet).
cl	(character()) Allowed subclasses.
no_untyped	(logical(1)) Are untyped Domains allowed?
must_bounded	(logical(1)) Only bounded Domains allowed?
no_deps	(logical(1)) Are dependencies allowed?

Value

The checked object, invisibly.

---

condition_test *Dependency Condition*

Description

Condition object, to specify the condition in a dependency.

Usage

```
condition_test(cond, x)

condition_as_string(cond, lhs_chr = "x")

Condition(rhs, condition_format_string)
```

Arguments

cond	(Condition) Condition to use
x	(any) Value to test
lhs_chr	(character(1)) Symbolic representation to use for <lhs> in the returned string.
rhs	(any) Right-hand-side of the condition.
condition_format_string	(character(1)) Format-string for representing the condition when pretty-printing in condition_as_string(). Should contain two %s, as it is used in an sprintf()-call with two further string values.

Functions

- `condition_test()`: Used internally. Tests whether a value satisfies a given condition. Vectorizes when `x` is atomic.
- `condition_as_string()`: Used internally. Returns a string that represents the condition for pretty printing, in the form `"<lhs> <relation> <rhs>"`, e.g. `"x == 3"` or `"param %in% {1, 2, 10}"`.

Currently implemented simple conditions

- `CondEqual(rhs)`
  Value must be equal to `rhs`.
- `CondAnyOf(rhs)`
  Value must be any value of `rhs`.

`default_values` *Extract Parameter Default Values*

Description

Extract parameter default values.

Usage

```
default_values(x, ...)

## S3 method for class 'ParamSet'
default_values(x, ...)
```

Arguments

`x`	(any) Object to extract default values from.
`...`	(any) Additional arguments.

Value

`list()`.

Design *Design of Configurations*

Description

A lightweight wrapper around a ParamSet and a `data.table::data.table()`, where the latter is a design of configurations produced from the former - e.g., by calling a `generate_design_grid()` or by sampling.

Public fields

`param_set` (ParamSet).

`data` (`data.table::data.table()`)
Stored data.

Methods

Public methods:

- `Design$new()`
- `Design$format()`
- `Design$print()`
- `Design$transpose()`
- `Design$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
Design$new(param_set, data, remove_dupl)
```

Arguments:

`param_set` (ParamSet).

`data` (`data.table::data.table()`)
Stored data.

`remove_dupl` (`logical(1)`)
Remove duplicates?

Method `format()`**:** Helper for print outputs.

Usage:

```
Design$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Design$print(...)
```

Arguments:

`...` (ignored).

Method `transpose()`**:** Converts `data` into a list of lists of row-configurations, possibly removes `NA` entries of inactive parameter values due to unsatisfied dependencies, and possibly calls the `trafo` function of the ParamSet.

Usage:

```
Design$transpose(filter_na = TRUE, trafo = TRUE)
```

Arguments:

`filter_na (logical(1))`
Should `NA` entries of inactive parameter values due to unsatisfied dependencies be removed?

`trafo (logical(1))`
Should the `trafo` function of the ParamSet be called?

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Design$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`Domain` *Domain: Parameter Range without an Id*

Description

A `Domain` object is a representation of a single dimension of a `ParamSet`. `Domain` objects are used to construct `ParamSet`s, either through the `ps()` short form, through the `ParamSet` constructor itself, or through the `ParamSet$search_space()` mechanism (see `to_tune()`). For each of the basic parameter classes (`"ParamInt"`, `"ParamDbl"`, `"ParamLgl"`, `"ParamFct"`, and `"ParamUty"`) there is a function constructing a `Domain` object (`p_int()`, `p_dbl()`, `p_lgl()`, `p_fct()`, `p_uty()`). They each have fitting construction arguments that control their bounds and behavior.

`Domain` objects are representations of parameter ranges and are intermediate objects to be used in short form constructions in `to_tune()` and `ps()`. Because of their nature, they should not be modified by the user, once constructed. The `Domain` object's internals are subject to change and should not be relied upon.

Usage

```
p_dbl(
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
```

```
  tolerance = sqrt(.Machine$double.eps),
  depends = NULL,
  trafo = NULL,
  logscale = FALSE,
  init,
  aggr = NULL,
  in_tune_fn = NULL,
  disable_in_tune = NULL
)

p_fct(
  levels,
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  depends = NULL,
  trafo = NULL,
  init,
  aggr = NULL,
  in_tune_fn = NULL,
  disable_in_tune = NULL
)

p_int(
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  tolerance = sqrt(.Machine$double.eps),
  depends = NULL,
  trafo = NULL,
  logscale = FALSE,
  init,
  aggr = NULL,
  in_tune_fn = NULL,
  disable_in_tune = NULL
)

p_lgl(
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  depends = NULL,
  trafo = NULL,
  init,
  aggr = NULL,
  in_tune_fn = NULL,
```

```
  disable_in_tune = NULL
)

p_uty(
  custom_check = NULL,
  special_vals = list(),
  default = NO_DEF,
  tags = character(),
  depends = NULL,
  trafo = NULL,
  repr = substitute(default),
  init,
  aggr = NULL,
  in_tune_fn = NULL,
  disable_in_tune = NULL
)
```

Arguments

lower	(numeric(1)) Lower bound, can be -Inf.
upper	(numeric(1)) Upper bound can be +Inf.
special_vals	(list()) Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.
default	(any) Default value. Can be from the domain of the parameter or an element of special_vals. Has value NO_DEF if no default exists. NULL can be a valid default. The value has no effect on ParamSet$values or the behavior of ParamSet$check(), $test() or $assert(). The default is intended to be used for documentation purposes. ‘
tags	(character()) Arbitrary tags to group and subset parameters. Some tags serve a special purpose: • "required" implies that the parameters has to be given when setting values in ParamSet.
tolerance	(numeric(1)) Initializes the $tolerance field that determines the
depends	(call | expression) An expression indicating a requirement for the parameter that will be constructed from this. Can be given as an expression (using quote()), or the expression can be entered directly and will be parsed using NSE (see examples). The expression may be of the form <Param> == <value> or <Param> %in% <values>, which will result in dependencies according to ParamSet$add_dep(on = "<Param>", cond = CondEqual(<val

or ParamSet$add_dep(on = "<Param>", cond = CondAnyOf(<values>)), respectively (see CondEqual, CondAnyOf). The expression may also contain multiple conditions separated by &&.

trafo (function)
Single argument function performing the transformation of a parameter. When the Domain is used to construct a ParamSet, this transformation will be applied to the corresponding parameter as part of the $trafo function.
Note that the trafo is *not* inherited by TuneTokens! Defining a parameter with e.g. p_dbl(..., trafo = ...) will *not* automatically give the to_tune() assigned to it a transformation. trafo only makes sense for ParamSets that get used as search spaces for optimization or tuning, it is not useful when defining domains or hyperparameter ranges of learning algorithms, because these do not use trafos.

logscale (logical(1))
Put numeric domains on a log scale. Default FALSE. Log-scale Domains represent parameter ranges where lower and upper bounds are logarithmized, and where a trafo is added that exponentiates sampled values to the original scale. This is *not* the same as setting trafo = exp, because logscale = TRUE will handle parameter bounds internally: a p_dbl(1, 10, logscale = TRUE) results in a parameter that has lower bound 0, upper bound log(10), and uses exp transformation on these. Therefore, the given bounds represent the bounds *after* the transformation. (see examples).
p_int() with logscale = TRUE results in a continuous parameter similar to p_dbl(), not an integer-valued parameter, with bounds log(max(lower, 0.5)) ... log(upper + 1) and a trafo similar to "as.integer(exp(x))" (with additional bounds correction). The lower bound is lifted to 0.5 if lower 0 to handle the lower == 0 case. The upper bound is increased to log(upper + 1) because the trafo would otherwise almost never generate a value of upper.
When logscale is TRUE, then upper bounds may be infinite, but lower bounds should be greater than 0 for p_dbl() or greater or equal 0 for p_int().
Note that "logscale" is *not* inherited by TuneTokens! Defining a parameter with p_dbl(... logscale = TRUE) will *not* automatically give the to_tune() assigned to it log-scale. logscale only makes sense for ParamSets that get used as search spaces for optimization or tuning, it is not useful when defining domains or hyperparameter ranges of learning algorithms, because these do not use trafos.
logscale happens on a natural (e == 2.718282...) basis. Be aware that using a different base (log10()/10^, log2()/2^) is completely equivalent and does not change the values being sampled after transformation.

init (any)
Initial value. When this is given, then the corresponding entry in ParamSet$values is initialized with this value upon construction.

aggr (function)
Default aggregation function for a parameter. Can only be given for parameters tagged with "internal_tuning". Function with one argument, which is a list of parameter values and that returns the aggregated parameter value.

in_tune_fn (function(domain, param_vals))
Function that converters a Domain object into a parameter value. Can only be

given for parameters tagged with "internal_tuning". This function should also assert that the parameters required to enable internal tuning for the given domain are set in param_vals (such as early_stopping_rounds for XGBoost).

disable_in_tune
(named list())
The parameter values that need to be set in the ParamSet to disable the internal tuning for the parameter. For XGBoost this would e.g. be list(early_stopping_rounds = NULL).

levels (character | atomic | list)
Allowed categorical values of the parameter. If this is not a character, then a trafo is generated that converts the names (if not given: as.character() of the values) of the levels argument to the values. This trafo is then performed *before* the function given as the trafo argument.

custom_check (function())
Custom function to check the feasibility. Function which checks the input. Must return 'TRUE' if the input is valid and a character(1) with the error message otherwise. This function should *not* throw an error. Defaults to NULL, which means that no check is performed.

repr (language)
Symbol to use to represent the value given in default. The deparse() of this object is used when printing the domain, in some cases.

Details

Although the levels values of a constructed p_fct() will always be character-valued, the p_fct function admits a levels argument that goes beyond this: Besides a character vector, any atomic vector or list (optionally named) may be given. (If the value is a list that is not named, the names are inferred using as.character() on the values.) The resulting Domain will correspond to a range of values given by the names of the levels argument with a trafo that maps the character names to the arbitrary values of the levels argument.

Value

A Domain object.

See Also

Other ParamSet construction helpers: ps(), to_tune()

Examples

```
params = ps(
  unbounded_integer = p_int(),
  bounded_double = p_dbl(0, 10),
  half_bounded_integer = p_dbl(1),
  half_bounded_double = p_dbl(upper = 1),
  double_with_trafo = p_dbl(-1, 1, trafo = exp),
  extra_double = p_dbl(0, 1, special_vals = list("xxx"), tags = "tagged"),
  factor_param = p_fct(c("a", "b", "c")),
```

```
  factor_param_with_implicit_trafo = p_fct(list(a = 1, b = 2, c = list()))
)
print(params)

params$trafo(list(
  bounded_double = 1,
  double_with_trafo = 1,
  factor_param = "c",
  factor_param_with_implicit_trafo = "c"
))

# logscale:
params = ps(x = p_dbl(1, 100, logscale = TRUE))

# The ParamSet has bounds log(1) .. log(100):
print(params)

# When generating a equidistant grid, it is equidistant within log values
grid = generate_design_grid(params, 3)
print(grid)

# But the values are on a log scale with desired bounds after trafo
print(grid$transpose())

# Integer parameters with logscale are `p_dbl()`s pre-trafo
params = ps(x = p_int(0, 10, logscale = TRUE))
print(params)

grid = generate_design_grid(params, 4)
print(grid)

# ... but get transformed to integers.
print(grid$transpose())


# internal tuning

param_set = ps(
 iters = p_int(0, Inf, tags = "internal_tuning", aggr = function(x) round(mean(unlist(x))),
    in_tune_fn = function(domain, param_vals) {
      stopifnot(domain$lower <= 1)
      stopifnot(param_vals$early_stopping == TRUE)
      domain$upper
    },
    disable_in_tune = list(early_stopping = FALSE)),
  early_stopping = p_lgl()
)
param_set$set_values(
  iters = to_tune(upper = 100, internal = TRUE),
  early_stopping = TRUE
)
param_set$convert_internal_search_space(param_set$search_space())
param_set$aggr_internal_tuned_values(
```

```
  list(iters = list(1, 2, 3))
)

param_set$disable_internal_tuning("iters")
param_set$values$early_stopping
```

`generate_design_grid` *Generate a Grid Design*

Description

Generate a grid with a specified resolution in the parameter space. The resolution for categorical parameters is ignored, these parameters always produce a grid over all their valid levels. For number params the endpoints of the params are always included in the grid.

Usage

```
generate_design_grid(param_set, resolution = NULL, param_resolutions = NULL)
```

Arguments

`param_set`	(ParamSet).
`resolution`	(`integer(1)`) Global resolution for all parameters.
`param_resolutions`	(named `integer()`) Resolution per Domain, named by parameter ID.

Value

Design.

See Also

Other generate_design: `generate_design_lhs()`, `generate_design_random()`, `generate_design_sobol()`

Examples

```
pset = ps(
  ratio = p_dbl(lower = 0, upper = 1),
  letters = p_fct(levels = letters[1:3])
)
generate_design_grid(pset, 10)
```

`generate_design_lhs` *Generate a Space-Filling LHS Design*

Description

Generate a space-filling design using Latin hypercube sampling. Dependent parameters whose constraints are unsatisfied generate `NA` entries in their respective columns.

Usage

```
generate_design_lhs(param_set, n, lhs_fun = NULL)
```

Arguments

`param_set`	(`ParamSet`).
`n`	(`integer(1)`) Number of points to sample.
`lhs_fun`	(`function(n, k)`) Function to use to generate a LHS sample, with n samples and k values per param. LHS functions are implemented in package **lhs**, default is to use `lhs::maximinLHS()`.

Value

`Design`.

See Also

Other generate_design: `generate_design_grid()`, `generate_design_random()`, `generate_design_sobol()`

Examples

```
pset = ps(
  ratio = p_dbl(lower = 0, upper = 1),
  letters = p_fct(levels = letters[1:3])
)

if (requireNamespace("lhs", quietly = TRUE)) {
  generate_design_lhs(pset, 10)
}
```

generate_design_random *Generate a Random Design*

Description

Generates a design with randomly drawn points. Internally uses SamplerUnif, hence, also works for ParamSets with dependencies. If dependencies do not hold, values are set to NA in the resulting data.table.

Usage

```
generate_design_random(param_set, n)
```

Arguments

param_set	(ParamSet).
n	(integer(1)) Number of points to draw randomly.

Value

Design.

See Also

Other generate_design: generate_design_grid(), generate_design_lhs(), generate_design_sobol()

Examples

```
pset = ps(
  ratio = p_dbl(lower = 0, upper = 1),
  letters = p_fct(levels = letters[1:3])
)
generate_design_random(pset, 10)
```

generate_design_sobol *Generate a Space-Filling Sobol Sequence Design*

Description

Generate a space-filling design using a Sobol sequence. Dependent parameters whose constraints are unsatisfied generate NA entries in their respective columns.

Uses spacefillr::generate_sobol_set.

Note that non determinism is achieved by sampling the seed argument via sample(.Machine$integer.max, size = 1L).

Usage

```
generate_design_sobol(param_set, n)
```

Arguments

param_set	(ParamSet).
n	(integer(1)) Number of points to sample.

Value

Design.

See Also

Other generate_design: generate_design_grid(), generate_design_lhs(), generate_design_random()

Examples

```
pset = ps(
  ratio = p_dbl(lower = 0, upper = 1),
  letters = p_fct(levels = letters[1:3])
)

if (requireNamespace("spacefillr", quietly = TRUE)) {
  generate_design_sobol(pset, 10)
}
```

NO_DEF *Extra data type for "no default value"*

Description

Special new data type for no-default. Not often needed by the end-user, mainly internal.

- NO_DEF: Singleton object for type, used in Domain when no default is given.
- is_nodefault(): Is an object the 'no default' object?

ParamSet *ParamSet*

Description

An object representing the space of possible parametrizations of a function or another object. ParamSets are used on the side of objects being parameterized, where they function as a configuration space determining the set of possible configurations accepted by these objects. They can also be used to specify search spaces for optimization, indicating the set of legal configurations to try out. It is often convenient to generate search spaces from configuration spaces, which can be done using the $search_space() method in combination with to_tune() / TuneToken objects.

Individual dimensions of a ParamSet are specified by Domain objects, created as p_dbl(), p_lgl() etc. The field $values can be used to store an active configuration or to partially fix some parameters to constant values – the precise effect can be determined by the object being parameterized.

Constructing a ParamSet can be done using ParamSet$new() in combination with a named list of Domain objects. This route is recommended when the set of dimensions (i.e. the members of this named list) is dynamically created, such as when the number of parameters is variable. ParamSets can also be created using the ps() shorthand, which is the recommended way when the set of parameters is fixed. In practice, the majority of cases where a ParamSet is created, the ps() should be used.

S3 methods and type converters

- as.data.table()
  ParamSet -> data.table::data.table()
  Compact representation as datatable. Col types are:
  - id: character
  - class: character
  - lower, upper: numeric
  - levels: list col, with NULL elements
  - nlevels: integer valued numeric
  - is_bounded: logical
  - special_vals: list col of list
  - default: list col
  - storage_type: character
  - tags: list col of character vectors

Public fields

assert_values (logical(1))
: Should values be checked for validity during assigment to active binding $values? Default is TRUE, only switch this off if you know what you are doing.

Active bindings

`data` (`data.table`) `data.table` representation of the `ParamSet`.

`values` (named `list()`)
Currently set / fixed parameter values. Settable, and feasibility of values will be checked when you set them. You do not have to set values for all parameters, but only for a subset. When you set values, all previously set values will be unset / removed.

`tags` (named `list()` of `character()`)
Can be used to group and subset parameters. Named with parameter IDs.

`params` (named `list()`)
`data.table` representing the combined `Domain` objects used to construct the `ParamSet`. Used for internal purposes. Its use by external code is deprecated.

`domains` (named `list` of `Domain`) List of `Domain` objects that could be used to initialize this `ParamSet`.

`extra_trafo` (`function(x, param_set)`)
Transformation function. Settable. User has to pass a `function(x)`, of the form
(named `list()`, ParamSet) -> named `list()`.
The function is responsible to transform a feasible configuration into another encoding, before potentially evaluating the configuration with the target algorithm. For the output, not many things have to hold. It needs to have unique names, and the target algorithm has to accept the configuration. For convenience, the self-paramset is also passed in, if you need some info from it (e.g. tags). Is NULL by default, and you can set it to NULL to switch the transformation off.

`constraint` (`function(x)`)
Constraint function. Settable. This function must evaluate a named `list()` of values and determine whether it satisfies constraints, returning a scalar `logical(1)` value.

`deps` (`data.table::data.table()`)
Table has cols `id` (`character(1)`) and `on` (`character(1)`) and `cond` (Condition). Lists all (direct) dependency parents of a param, through parameter IDs. Internally created by a call to `add_dep`. Settable, if you want to remove dependencies or perform other changes.

`length` (`integer(1)`)
Number of contained parameters.

`is_empty` (`logical(1)`)
Is the `ParamSet` empty? Named with parameter IDs.

`has_trafo` (`logical(1)`)
Whether a `trafo` function is present, in parameters or in `extra_trafo`.

`has_extra_trafo` (`logical(1)`)
Whether `extra_trafo` is set.

`has_deps` (`logical(1)`)
Whether the parameter dependencies are present

`has_constraint` (`logical(1)`)
Whether parameter constraint is set.

`all_numeric` (`logical(1)`)
Is `TRUE` if all parameters are `p_dbl()` or `p_int()`.

`all_categorical` (`logical(1)`)
: Is `TRUE` if all parameters are `p_fct()` and `p_lgl()`.

`all_bounded` (`logical(1)`)
: Is `TRUE` if all parameters are bounded.

`class` (named `character()`)
: Classes of contained parameters. Named with parameter IDs.

`lower` (named `double()`)
: Lower bounds of numeric parameters (`NA` for non-numerics). Named with parameter IDs.

`upper` (named `double()`)
: Upper bounds of numeric parameters (`NA` for non-numerics). Named with parameter IDs.

`levels` (named `list()` of `character`)
: Allowed levels of categorical parameters (`NULL` for non-categoricals). Named with parameter IDs.

`storage_type` (`character()`)
: Data types of parameters when stored in tables. Named with parameter IDs.

`special_vals` (named `list()` of `list()`)
: Special values for all parameters. Named with parameter IDs.

`default` (named `list()`)
: Default values of all parameters. If no default exists, element is not present. Named with parameter IDs.

`has_trafo_param` (`logical()`)
: Whether `trafo` is set for any parameter.

`is_logscale` (`logical()`)
: Whether `trafo` was set to `logscale` during construction.
Note that this only refers to the `logscale` flag set during construction, e.g. `p_dbl(logscale = TRUE)`. If the parameter was set to logscale manually, e.g. through `p_dbl(trafo = exp)`, this `is_logscale` will be `FALSE`.

`nlevels` (named `integer()`)
: Number of distinct levels of parameters. `Inf` for double parameters or unbounded integer parameters. Named with param IDs.

`is_number` (named `logical()`)
: Whether parameter is `p_dbl()` or `p_int()`. Named with parameter IDs.

`is_categ` (named `logical()`)
: Whether parameter is `p_fct()` or `p_lgl()`. Named with parameter IDs.

`is_bounded` (named `logical()`)
: Whether parameters have finite bounds. Named with parameter IDs.

Methods

Public methods:

- `ParamSet$new()`
- `ParamSet$ids()`
- `ParamSet$get_values()`
- `ParamSet$set_values()`

- ParamSet$trafo()
- ParamSet$aggr_internal_tuned_values()
- ParamSet$disable_internal_tuning()
- ParamSet$convert_internal_search_space()
- ParamSet$test_constraint()
- ParamSet$test_constraint_dt()
- ParamSet$check()
- ParamSet$check_dependencies()
- ParamSet$test()
- ParamSet$assert()
- ParamSet$check_dt()
- ParamSet$test_dt()
- ParamSet$assert_dt()
- ParamSet$qunif()
- ParamSet$get_domain()
- ParamSet$subset()
- ParamSet$subspaces()
- ParamSet$flatten()
- ParamSet$search_space()
- ParamSet$add_dep()
- ParamSet$format()
- ParamSet$print()
- ParamSet$clone()

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ParamSet$new(params = named_list(), allow_dangling_dependencies = FALSE)
```

Arguments:

`params` (named `list()`)
: List of Domain, named with their respective ID.

`allow_dangling_dependencies` (`character(1)`)
: Whether dependencies depending on parameters that are not present should be allowed. A parameter `x` having `depends = y == 0` if `y` is not present would usually throw an error, but if dangling dependencies are allowed, the dependency is added regardless. This is mainly for internal use.

Method `ids()`**:** Retrieves IDs of contained parameters based on some filter criteria selections, `NULL` means no restriction. Only returns IDs of parameters that satisfy all conditions.

Usage:

```
ParamSet$ids(class = NULL, tags = NULL, any_tags = NULL)
```

Arguments:

`class` (`character()`)
: Typically a subset of `"ParamDbl"`, `"ParamInt"`, `"ParamFct"`, `"ParamLgl"`, `"ParamUty"`. Other classes are possible if implemented by 3rd party packages. Return only IDs of dimensions with the given class.

`tags` (`character()`). Return only IDs of dimensions that have *all* tags given in this argument.

`any_tags` (`character()`). Return only IDs of dimensions that have at least one of the tags given in this argument.

Returns: `character()`.

Method `get_values()`**:** Retrieves parameter values based on some selections, `NULL` means no restriction and is equivalent to `$values`. Only returns values of parameters that satisfy all conditions.

Usage:

```
ParamSet$get_values(
  class = NULL,
  tags = NULL,
  any_tags = NULL,
  type = "with_token",
  check_required = TRUE,
  remove_dependencies = TRUE
)
```

Arguments:

`class` (`character()`). See `$ids()`.

`tags` (`character()`). See `$ids()`.

`any_tags` (`character()`). See `$ids()`.

`type` (`character(1)`)
Return values `"with_token"` (i.e. all values),

`check_required` (`logical(1)`)
Check if all required parameters are set?

`remove_dependencies` (`logical(1)`)
If `TRUE`, set values with dependencies that are not fulfilled to `NULL`.

Returns: Named `list()`.

Method `set_values()`**:** Allows to to modify (and overwrite) or replace the parameter values. Per default already set values are being kept unless new values are being provided.

Usage:

```
ParamSet$set_values(..., .values = list(), .insert = TRUE)
```

Arguments:

`...` (any)
Named parameter values.

`.values` (named `list()`)
Named list with parameter values. Names must not already appear in `...`.

`.insert` (`logical(1)`)
Whether to insert the values (old values are being kept, if not overwritten), or to replace all values. Default is TRUE.

Method `trafo()`**:** Perform transformation specified by the `trafo` of `Domain` objects, as well as the `$extra_trafo` field.

Usage:

```
ParamSet$trafo(x, param_set = self)
```

Arguments:

`x` (named `list()` | `data.frame`)
The value(s) to be transformed.

`param_set` (`ParamSet`)
Passed to `extra_trafo()`. Note that the `extra_trafo` of `self` is used, not the `extra_trafo` of the `ParamSet` given in the `param_set` argument. In almost all cases, the default `param_set = self` should be used.

Method `aggr_internal_tuned_values()`**:** Aggregate parameter values according to their aggregation rules.

Usage:

```
ParamSet$aggr_internal_tuned_values(x)
```

Arguments:

`x` (named `list()` of `list()`s)
The value(s) to be aggregated. Names are parameter values. The aggregation function is selected based on the parameter.

Returns: (named `list()`)

Method `disable_internal_tuning()`**:** Set the parameter values so that internal tuning for the selected parameters is disabled.

Usage:

```
ParamSet$disable_internal_tuning(ids)
```

Arguments:

`ids` (`character()`)
The ids of the parameters for which to disable internal tuning.

Returns: `Self`

Method `convert_internal_search_space()`**:** Convert all parameters from the search space to parameter values using the transformation given by `in_tune_fn`.

Usage:

```
ParamSet$convert_internal_search_space(search_space)
```

Arguments:

`search_space` (`ParamSet`)
The internal search space.

Returns: (named `list()`)

Method `test_constraint()`**:** **checkmate**-like test-function. Takes a named list. Return `FALSE` if the given `$constraint` is not satisfied, `TRUE` otherwise. Note this is different from satisfying the bounds or types given by the `ParamSet` itself: If `x` does not satisfy these, an error will be thrown, given that `assert_value` is `TRUE`.

Usage:

```
ParamSet$test_constraint(x, assert_value = TRUE)
```

Arguments:

`x` (named `list()`)
The value to test.

`assert_value (logical(1))`
Whether to verify that `x` satisfies the bounds and types given by this `ParamSet`. Should be `TRUE` unless this was already checked before.

Returns: `logical(1)`: Whether `x` satisfies the `$constraint`.

Method `test_constraint_dt()`**: checkmate**-like test-function. Takes a `data.table`. For each row, return `FALSE` if the given `$constraint` is not satisfied, `TRUE` otherwise. Note this is different from satisfying the bounds or types given by the `ParamSet` itself: If `x` does not satisfy these, an error will be thrown, given that `assert_value` is `TRUE`.

Usage:

```
ParamSet$test_constraint_dt(x, assert_value = TRUE)
```

Arguments:

`x` (`data.table`)
The values to test.

`assert_value (logical(1))`
Whether to verify that `x` satisfies the bounds and types given by this `ParamSet`. Should be `TRUE` unless this was already checked before.

Returns: `logical`: For each row in `x`, whether it satisfies the `$constraint`.

Method `check()`**: checkmate**-like check-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of `x`. Constraints and dependencies are not checked when `check_strict` is `FALSE`.

Usage:

```
ParamSet$check(xs, check_strict = TRUE, sanitize = FALSE)
```

Arguments:

`xs` (named `list()`).

`check_strict (logical(1))`
Whether to check that constraints and dependencies are satisfied.

`sanitize (logical(1))`
Whether to move values that are slightly outside bounds to valid values. These values are accepted independent of `sanitize` (depending on the `tolerance` arguments of `p_dbl()` and `p_int()`). If `sanitize` is `TRUE`, the additional effect is that, should checks pass, the sanitized values of `xs` are added to the result as attribute `"sanitized"`.

Returns: If successful `TRUE`, if not a string with an error message.

Method `check_dependencies()`**: checkmate**-like check-function. Takes a named list. Checks that all individual param dependencies are satisfied.

Usage:

```
ParamSet$check_dependencies(xs)
```

Arguments:

`xs` (named `list()`).

Returns: If successful `TRUE`, if not a string with an error message.

Method `test()`**: checkmate**-like test-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of `x`. Constraints and dependencies are not checked when `check_strict` is `FALSE`.

Usage:

```
ParamSet$test(xs, check_strict = TRUE)
```

Arguments:

`xs` (named `list()`).

`check_strict (logical(1))`
Whether to check that constraints and dependencies are satisfied.

Returns: If successful `TRUE`, if not `FALSE`.

Method `assert()`**: checkmate**-like assert-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of `x`. Constraints and dependencies are not checked when `check_strict` is `FALSE`.

Usage:

```
ParamSet$assert(
  xs,
  check_strict = TRUE,
  .var.name = vname(xs),
  sanitize = FALSE
)
```

Arguments:

`xs` (named `list()`).

`check_strict (logical(1))`
Whether to check that constraints and dependencies are satisfied.

`.var.name (character(1))`
Name of the checked object to print in error messages.
Defaults to the heuristic implemented in vname.

`sanitize (logical(1))`
Whether to move values that are slightly outside bounds to valid values. These values are accepted independent of `sanitize` (depending on the `tolerance` arguments of `p_dbl()` and `p_int()`). If `sanitize` is `TRUE`, the additional effect is that `xs` is converted to within bounds.

Returns: If successful `xs` invisibly, if not an error message.

Method `check_dt()`**: checkmate**-like check-function. Takes a data.table::data.table where rows are points and columns are parameters. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of `x`. Constraints and dependencies are not checked when `check_strict` is `FALSE`.

Usage:

```
ParamSet$check_dt(xdt, check_strict = TRUE)
```

Arguments:

`xdt` (data.table::data.table | `data.frame()`).

`check_strict` (`logical(1)`)
Whether to check that constraints and dependencies are satisfied.

Returns: If successful `TRUE`, if not a string with the error message.

Method `test_dt()`**:** **checkmate**-like test-function (s. `$check_dt()`).

Usage:

```
ParamSet$test_dt(xdt, check_strict = TRUE)
```

Arguments:

`xdt` (data.table::data.table).

`check_strict` (`logical(1)`)
Whether to check that constraints and dependencies are satisfied.

Returns: If successful `TRUE`, if not `FALSE`.

Method `assert_dt()`**:** **checkmate**-like assert-function (s. `$check_dt()`).

Usage:

```
ParamSet$assert_dt(xdt, check_strict = TRUE, .var.name = vname(xdt))
```

Arguments:

`xdt` (data.table::data.table).

`check_strict` (`logical(1)`)
Whether to check that constraints and dependencies are satisfied.

`.var.name` (`character(1)`)
Name of the checked object to print in error messages.
Defaults to the heuristic implemented in vname.

Returns: If successful `xs` invisibly, if not, an error is generated.

Method `qunif()`**:** Map a `matrix` or `data.frame` of values between 0 and 1 to proportional values inside the feasible intervals of individual parameters.

Usage:

```
ParamSet$qunif(x)
```

Arguments:

`x` (`matrix` | `data.frame`)
Values to map. Column names must be a subset of the names of parameters.

Returns: `data.table`.

Method `get_domain()`**:** get the `Domain` object that could be used to create a given parameter.

Usage:

```
ParamSet$get_domain(id)
```

Arguments:

`id` (`character(1)`).

Returns: `Domain`.

Method `subset()`**:** Create a new `ParamSet` restricted to the passed IDs.

Usage:

```
ParamSet$subset(
  ids,
  allow_dangling_dependencies = FALSE,
  keep_constraint = TRUE
)
```

Arguments:

`ids` (`character()`).

`allow_dangling_dependencies` (`logical(1)`)
Whether to allow subsets that cut across parameter dependencies. Dependencies that point to dropped parameters are kept (but will be "dangling", i.e. their `"on"` will not be present).

`keep_constraint` (`logical(1)`)
Whether to keep the `$constraint` function.

Returns: `ParamSet`.

Method `subspaces()`**:** Create new one-dimensional `ParamSets` for each dimension.

Usage:

```
ParamSet$subspaces(ids = private$.params$id)
```

Arguments:

`ids` (`character()`)
IDs for which to create `ParamSets`. Defaults to all IDs.

Returns: named `list()` of `ParamSet`.

Method `flatten()`**:** Create a `ParamSet` from this object, even if this object itself is not a `ParamSet` but e.g. a `ParamSetCollection`.

Usage:

```
ParamSet$flatten()
```

Method `search_space()`**:** Construct a `ParamSet` to tune over. Constructed from `TuneToken` in `$values`, see `to_tune()`.

Usage:

```
ParamSet$search_space(values = self$values)
```

Arguments:

`values` (named `list`): optional named list of `TuneToken` objects to convert, in place of `$values`.

Method `add_dep()`**:** Adds a dependency to this set, so that param `id` now depends on param on.

Usage:

```
ParamSet$add_dep(id, on, cond, allow_dangling_dependencies = FALSE)
```

Arguments:

`id` (`character(1)`).
`on` (`character(1)`).
`cond` (Condition).
`allow_dangling_dependencies` (`logical(1)`): Whether to allow dependencies on parameters that are not present.

Method `format()`**:** Helper for print outputs.

Usage:

```
ParamSet$format()
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
ParamSet$print(
  ...,
 hide_cols = c("levels", "is_bounded", "special_vals", "tags", "storage_type")
)
```

Arguments:

`...` (ignored).
`hide_cols` (`character()`)
Which fields should not be printed? Default is `"levels"`, `"is_bounded"`, `"special_vals"`, `"tags"`, and `"storage_type"`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ParamSet$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
pset = ParamSet$new(
  params = list(
    d = p_dbl(lower = -5, upper = 5, default = 0, trafo = function(x) 2^x),
    f = p_fct(levels = letters[1:3])
  )
)

# alternative, recommended way of construction in this case since the
# parameter list is not dynamic:
pset = ps(
  d = p_dbl(lower = -5, upper = 5, default = 0, trafo = function(x) 2^x),
  f = p_fct(levels = letters[1:3])
)

pset$check(list(d = 2.1, f = "a"))

pset$check(list(d = 2.1, f = "d"))
```

ParamSetCollection *ParamSetCollection*

Description

A collection of multiple ParamSet objects.

- The collection is basically a light-weight wrapper / container around references to multiple sets.
- In order to ensure unique param names, every param in the collection is referred to with "<set_id>.<param_id>", where <set_id> is the name of the entry a given ParamSet in the named list given during construction. Parameters from ParamSet with empty (i.e. "") set_id are referenced directly. Multiple ParamSets with set_id "" can be combined, but their parameter names may not overlap to avoid name clashes.
- When you either ask for 'values' or set them, the operation is delegated to the individual, contained ParamSet references. The collection itself does not maintain a values state. This also implies that if you directly change values in one of the referenced sets, this change is reflected in the collection.
- Dependencies: It is possible to currently handle dependencies
  - regarding parameters inside of the same set - in this case simply add the dependency to the set, best before adding the set to the collection
  - across sets, where a param from one set depends on the state of a param from another set - in this case add call add_dep on the collection.

  If you call deps on the collection, you are returned a complete table of dependencies, from sets and across sets.

Super class

paradox::ParamSet -> ParamSetCollection

Active bindings

deps (data.table::data.table())
: Table has cols id (character(1)) and on (character(1)) and cond (Condition). Lists all (direct) dependency parents of a param, through parameter IDs. Internally created by a call to add_dep. Settable, if you want to remove dependencies or perform other changes.

extra_trafo (function(x, param_set))
: Transformation function. Settable. User has to pass a function(x), of the form (named list(), ParamSet) -> named list().
The function is responsible to transform a feasible configuration into another encoding, before potentially evaluating the configuration with the target algorithm. For the output, not many things have to hold. It needs to have unique names, and the target algorithm has to accept the configuration. For convenience, the self-paramset is also passed in, if you need some info from it (e.g. tags). Is NULL by default, and you can set it to NULL to switch the transformation off.

`constraint` (`function(x)`)
Constraint function. Settable. This function must evaluate a named `list()` of values and determine whether it satisfies constraints, returning a scalar `logical(1)` value.

`sets` (named `list()`)
Read-only `list` of of `ParamSet`s contained in this `ParamSetCollection`. This field provides direct references to the `ParamSet` objects.

Methods

Public methods:

- `ParamSetCollection$new()`
- `ParamSetCollection$add()`
- `ParamSetCollection$disable_internal_tuning()`
- `ParamSetCollection$convert_internal_search_space()`
- `ParamSetCollection$flatten()`
- `ParamSetCollection$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
ParamSetCollection$new(sets, tag_sets = FALSE, tag_params = FALSE)
```

Arguments:

`sets` (named `list()` of ParamSet)
ParamSet objects are not cloned. Names are used as "set_id" for the naming scheme of delegated parameters.

`tag_sets` (`logical(1)`)
Whether to add tags of the form `"set_<set_id>"` to each parameter originating from a given `ParamSet` given with name `<set_id>`.

`tag_params` (`logical(1)`)
Whether to add tags of the form `"param_<param_id>"` to each parameter with original ID `<param_id>`.

Method `add()`**:** Adds a `ParamSet` to this collection.

Usage:

```
ParamSetCollection$add(p, n = "", tag_sets = FALSE, tag_params = FALSE)
```

Arguments:

`p` (ParamSet).

`n` (`character(1)`)
Name to use. Default `""`.

`tag_sets` (`logical(1)`)
Whether to add tags of the form `"set_<n>"` to the newly added parameters.

`tag_params` (`logical(1)`)
Whether to add tags of the form `"param_<param_id>"` to each parameter with original ID `<param_id>`.

Method `disable_internal_tuning()`**:** Set the parameter values so that internal tuning for the selected parameters is disabled.

Usage:

```
ParamSetCollection$disable_internal_tuning(ids)
```

Arguments:

`ids` (`character()`)
The ids of the parameters for which to disable internal tuning.

Returns: `Self`

Method `convert_internal_search_space()`**:** Convert all parameters from the search space to parameter values using the transformation given by `in_tune_fn`.

Usage:

```
ParamSetCollection$convert_internal_search_space(search_space)
```

Arguments:

`search_space` (`ParamSet`)
The internal search space.

Returns: (named `list()`)

Method `flatten()`**:** Create a `ParamSet` from this `ParamSetCollection`.

Usage:

```
ParamSetCollection$flatten()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ParamSetCollection$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`ps` *Construct a ParamSet using Short Forms*

Description

The `ps()` short form constructor uses `Domain` objects (`p_dbl`, `p_fct`, ...) to construct `ParamSet`s in a succinct and readable way.

For more specifics also see the documentation of `Domain`.

Usage

```
ps(
  ...,
  .extra_trafo = NULL,
  .constraint = NULL,
  .allow_dangling_dependencies = FALSE
)
```

Arguments

`...`	(Domain) Named arguments of Domain objects. The ParamSet will be constructed of the given Domains, The names of the arguments will be used as `$id()` in the resulting ParamSet.
`.extra_trafo`	(`function(x, param_set)`) Transformation to set the resulting ParamSet's `$trafo` value to. This is in addition to any `trafo` of Domain objects given in `...`, and will be run *after* transformations of individual parameters were performed.
`.constraint`	(`function(x)`) Constraint function. When given, this function must evaluate a named `list()` of values and determine whether it satisfies constraints, returning a scalar `logical(1)` value.
`.allow_dangling_dependencies`	(`logical`) Whether dependencies depending on parameters that are not present should be allowed. A parameter `x` having `depends = y == 0` if `y` is not present in the `ps()` call would usually throw an error, but if dangling dependencies are allowed, the dependency is added regardless. This is usually a bad idea and mainly for internal use. Dependencies between ParamSets when using `to_tune()` can be realized using this.

Value

A ParamSet object.

See Also

Other ParamSet construction helpers: `Domain()`, `to_tune()`

Examples

```
pars = ps(
  a = p_int(0, 10),
  b = p_int(upper = 20),
  c = p_dbl(),
  e = p_fct(letters[1:3]),
  f = p_uty(custom_check = checkmate::check_function)
)
print(pars)

pars = ps(
  a = p_dbl(0, 1, trafo = exp),
  b = p_dbl(0, 1, trafo = exp),
  .extra_trafo = function(x, ps) {
    x$c <- x$a + x$b
    x
  }
)
```

```
# See how the addition happens after exp()ing:
pars$trafo(list(a = 0, b = 0))

pars$values = list(
  a = to_tune(ps(x = p_int(0, 1),
    .extra_trafo = function(x, param_set) list(a = x$x)
  )),
  # make 'y' depend on 'x', but they are defined in different ParamSets
  # Therefore we need to allow dangling dependencies here.
  b = to_tune(ps(y = p_int(0, 1, depends = x == 1),
    .extra_trafo = function(x, param_set) list(b = x$y),
    .allow_dangling_dependencies = TRUE
  ))
)

pars$search_space()
```

---

psc *Create a ParamSet Collection*

Description

Creates a ParamSetCollection.

Usage

```
psc(...)
```

Arguments

...	(any) The ParamSets from which to create the collection.

---

ps_replicate *Create a ParamSet by Repeating a Given ParamSet*

Description

Repeat a ParamSet a given number of times and thus create a larger ParamSet. By default, the resulting parameters are prefixed with the string "repX.", where X counts up from 1. It is also possible to tag paramc easier.

Usage

```
ps_replicate(
  set,
  times = length(prefixes),
  prefixes = sprintf("rep%s", seq_len(times)),
  tag_sets = FALSE,
  tag_params = FALSE
)
```

Arguments

set	(ParamSet) ParamSet to use as template.
times	(integer(1)) Number of times to repeat set. Should not be given if prefixes is provided.
prefixes	(character) A character vector indicating the prefixes to use for each repetition of set. If this is given, times is inferred from length(prefixes) and should not be given separately. If times is given, this defaults to "repX", with X counting up from 1.
tag_sets	(logical(1)) Whether to add a tag of the form "set_<prefixes[[i]]>" to each parameter in the result, indicating the repetition each parameter belongs to.
tag_params	(logical(1)) Whether to add a tag of the form "param_<id>" to each parameter in the result, indicating the original parameter ID inside set.

Examples

```
pset = ps(
  i = p_int(),
  z = p_lgl()
)

ps_replicate(pset, 3)

ps_replicate(pset, prefixes = c("first", "last"))

pset$values = list(i = 1, z = FALSE)

psr = ps_replicate(pset, 2, tag_sets = TRUE, tag_params = TRUE)

# observe the effect of tag_sets, tag_params:
psr$tags

# note that values are repeated as well
psr$values

psr$set_values(rep1.i = 10, rep2.z = TRUE)
```

```
psr$values

# use `any_tags` to get subset of values.
# `any_tags = ` is preferable to `tags = `, since parameters
# could also have other tags. `tags = ` would require the
# selected params to have the given tags exclusively.

# get all values associated with the original parameter `i`
psr$get_values(any_tags = "param_i")

# get all values associated with the first repetition "rep1"
psr$get_values(any_tags = "set_rep1")
```

`ps_union` *Create a ParamSet from a list of ParamSets*

Description

This emulates `ParamSetCollection$new(sets)`, except that the result is a flat `ParamSet`, not a `ParamSetCollection`. The resulting object is decoupled from the input `ParamSet` objects: Unlike `ParamSetCollection`, changing `$values` of the resulting object will not change the input `ParamSet` `$values` by reference.

This emulates `ParamSetCollection$new(sets)`, which in particular means that the resulting `ParamSet` has all the `Domain`s from the input `sets`, but some `$id`s are changed: If the `ParamSet` is given in `sets` with a name, then the `Domain`s will have their `<id>` changed to `<name in "sets">.<id>`. This is also reflected in deps.

The `c()` operator, applied to `ParamSet`s, is a synony for `ps_union()`.

Usage

```
ps_union(sets, tag_sets = FALSE, tag_params = FALSE)
```

Arguments

`sets`	(`list` of `ParamSet`) This may be a named list, in which case non-empty names are prefixed to parameters in the corresponding `ParamSet`.
`tag_sets`	(`logical(1)`) Whether to add tags of the form `"set_<set_id>"` to each parameter originating from a given `ParamSet` given with name `<name in "sets">`.
`tag_params`	(`logical(1)`) Whether to add tags of the form `"param_<param_id>"` to each parameter with original ID `<param_id>`.

Examples

```
ps1 = ps(x = p_dbl())
ps1$values = list(x = 1)

ps2 = ps(y = p_lgl())

pu = ps_union(list(ps1, ps2))
# same as:
pu = c(ps1, ps2)

pu

pu$values

pu$values$x = 2
pu$values

# p1 is unchanged:
ps1$values

# Prefixes automatically created for named elements.
# This allows repeating components.
pu2 = c(one = ps1, two = ps1, ps2)
pu2

pu2$values
```

`Sampler` *Sampler Class*

Description

This is the abstract base class for sampling objects like Sampler1D, SamplerHierarchical or SamplerJointIndep.

Public fields

`param_set` (ParamSet)
: Domain / support of the distribution we want to sample from.

Methods

Public methods:

- `Sampler$new()`
- `Sampler$sample()`
- `Sampler$format()`
- `Sampler$print()`

- Sampler$clone()

Method `new()`**:** Creates a new instance of this R6 class.
Note that this object is typically constructed via derived classes, e.g., Sampler1D.

Usage:

```
Sampler$new(param_set)
```

Arguments:

`param_set` (ParamSet)
The ParamSet to associated with this Sampler.

Method `sample()`**:** Sample `n` values from the distribution.

Usage:

```
Sampler$sample(n)
```

Arguments:

`n` (integer(1)).

Returns: Design.

Method `format()`**:** Helper for print outputs.

Usage:

```
Sampler$format(...)
```

Arguments:

`...` (ignored).

Method `print()`**:** Printer.

Usage:

```
Sampler$print(...)
```

Arguments:

`...` (ignored).

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Sampler$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: Sampler1D, Sampler1DCateg, Sampler1DNormal, Sampler1DRfun, Sampler1DUnif, SamplerHierarchical, SamplerJointIndep, SamplerUnif

Sampler1D *Sampler1D Class*

Description

1D sampler, abstract base class for Sampler like Sampler1DUnif, Sampler1DRfun, Sampler1DCateg and Sampler1DNormal.

Super class

`paradox::Sampler` -> `Sampler1D`

Active bindings

`param` (`ParamSet`)
Returns the one-dimensional `ParamSet` that is sampled from.

Methods

Public methods:

- `Sampler1D$new()`
- `Sampler1D$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Note that this object is typically constructed via derived classes, e.g., `Sampler1DUnif`.

Usage:

```
Sampler1D$new(param)
```

Arguments:

`param` (`ParamSet`)
Domain / support of the distribution we want to sample from. Must be one-dimensional.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Sampler1D$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: `Sampler`, `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DRfun`, `Sampler1DUnif`, `SamplerHierarchical`, `SamplerJointIndep`, `SamplerUnif`

Sampler1DCateg *Sampler1DCateg Class*

Description

Sampling from a discrete distribution, for a ParamSet containing a single p_fct() or p_lgl().

Super classes

paradox::Sampler -> paradox::Sampler1D -> Sampler1DCateg

Public fields

prob (numeric() | NULL)
Numeric vector of param$nlevels probabilities.

Methods

Public methods:

- Sampler1DCateg$new()
- Sampler1DCateg$clone()

Method new(): Creates a new instance of this R6 class.

Usage:

```
Sampler1DCateg$new(param, prob = NULL)
```

Arguments:

param (ParamSet)
Domain / support of the distribution we want to sample from. Must be one-dimensional.

prob (numeric() | NULL)
Numeric vector of param$nlevels probabilities, which is uniform by default.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
Sampler1DCateg$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: Sampler, Sampler1D, Sampler1DNormal, Sampler1DRfun, Sampler1DUnif, SamplerHierarchical, SamplerJointIndep, SamplerUnif

Sampler1DNormal *Sampler1DNormal Class*

Description

Normal sampling (potentially truncated) for p_dbl().

Super classes

paradox::Sampler -> paradox::Sampler1D -> paradox::Sampler1DRfun -> Sampler1DNormal

Active bindings

mean (numeric(1))
: Mean parameter of the normal distribution.

sd (numeric(1))
: SD parameter of the normal distribution.

Methods

Public methods:

- Sampler1DNormal$new()
- Sampler1DNormal$clone()

Method new(): Creates a new instance of this R6 class.

Usage:

```
Sampler1DNormal$new(param, mean = NULL, sd = NULL)
```

Arguments:

param (ParamSet)
: Domain / support of the distribution we want to sample from. Must be one-dimensional.

mean (numeric(1))
: Mean parameter of the normal distribution. Default is mean(c(param$lower, param$upper).

sd (numeric(1))
: SD parameter of the normal distribution. Default is (param$upper - param$lower)/4.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
Sampler1DNormal$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: Sampler, Sampler1D, Sampler1DCateg, Sampler1DRfun, Sampler1DUnif, SamplerHierarchical, SamplerJointIndep, SamplerUnif

Sampler1DRfun *Sampler1DRfun Class*

Description

Arbitrary sampling from 1D RNG functions from R.

Super classes

`paradox::Sampler` -> `paradox::Sampler1D` -> `Sampler1DRfun`

Public fields

`rfun (function())`
Random number generator function.

`trunc (logical(1))`
`TRUE` enables naive rejection sampling, so we stay inside of [lower, upper].

Methods

Public methods:

- `Sampler1DRfun$new()`
- `Sampler1DRfun$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
Sampler1DRfun$new(param, rfun, trunc = TRUE)
```

Arguments:

`param (ParamSet)`
Domain / support of the distribution we want to sample from. Must be one-dimensional.

`rfun (function())`
Random number generator function, e.g. `rexp` to sample from exponential distribution.

`trunc (logical(1))`
`TRUE` enables naive rejection sampling, so we stay inside of [lower, upper].

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Sampler1DRfun$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: `Sampler`, `Sampler1D`, `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DUnif`, `SamplerHierarchical`, `SamplerJointIndep`, `SamplerUnif`

Sampler1DUnif *Sampler1DUnif Class*

Description

Uniform random sampler for arbitrary (bounded) parameters.

Super classes

`paradox::Sampler` -> `paradox::Sampler1D` -> `Sampler1DUnif`

Methods

Public methods:

- `Sampler1DUnif$new()`
- `Sampler1DUnif$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
Sampler1DUnif$new(param)
```

Arguments:

`param` (`ParamSet`)
Domain / support of the distribution we want to sample from. Must be one-dimensional.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Sampler1DUnif$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: `Sampler`, `Sampler1D`, `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DRfun`, `SamplerHierarchical`, `SamplerJointIndep`, `SamplerUnif`

SamplerHierarchical *SamplerHierarchical Class*

Description

Hierarchical sampling for arbitrary param sets with dependencies, where the user specifies 1D samplers per param. Dependencies are topologically sorted, parameters are then sampled in topological order, and if dependencies do not hold, values are set to NA in the resulting data.table.

Super class

paradox::Sampler -> SamplerHierarchical

Public fields

samplers (list())
: List of Sampler1D objects that gives a Sampler for each dimension in the param_set.

Methods

Public methods:

- SamplerHierarchical$new()
- SamplerHierarchical$clone()

Method new(): Creates a new instance of this R6 class.

Usage:

```
SamplerHierarchical$new(param_set, samplers)
```

Arguments:

param_set (ParamSet)
: The ParamSet to associated with this SamplerHierarchical.

samplers (list())
: List of Sampler1D objects that gives a Sampler for each dimension in the param_set.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
SamplerHierarchical$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: Sampler, Sampler1D, Sampler1DCateg, Sampler1DNormal, Sampler1DRfun, Sampler1DUnif, SamplerJointIndep, SamplerUnif

SamplerJointIndep *SamplerJointIndep Class*

Description

Create joint, independent sampler out of multiple other samplers.

Super class

`paradox::Sampler` -> `SamplerJointIndep`

Public fields

`samplers (list())`
List of `Sampler` objects.

Methods

Public methods:

- `SamplerJointIndep$new()`
- `SamplerJointIndep$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
SamplerJointIndep$new(samplers)
```

Arguments:

`samplers (list())`
List of `Sampler` objects.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SamplerJointIndep$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: `Sampler`, `Sampler1D`, `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DRfun`, `Sampler1DUnif`, `SamplerHierarchical`, `SamplerUnif`

SamplerUnif *SamplerUnif Class*

Description

Uniform random sampling for an arbitrary (bounded) ParamSet. Constructs 1 uniform sampler per parameter, then passes them to SamplerHierarchical. Hence, also works for ParamSets sets with dependencies.

Super classes

`paradox::Sampler` -> `paradox::SamplerHierarchical` -> `SamplerUnif`

Methods

Public methods:

- `SamplerUnif$new()`
- `SamplerUnif$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
SamplerUnif$new(param_set)
```

Arguments:

`param_set` (`ParamSet`)
The `ParamSet` to associated with this `SamplerUnif`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SamplerUnif$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: `Sampler`, `Sampler1D`, `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DRfun`, `Sampler1DUnif`, `SamplerHierarchical`, `SamplerJointIndep`

`to_tune` *Indicate that a Parameter Value should be Tuned*

Description

`to_tune()` creates a `TuneToken` object which can be assigned to the `$values` slot of a `ParamSet` as an alternative to a concrete value. This indicates that the value is not given directly but should be tuned using **bbotk** or **mlr3tuning**. If the thus parameterized object is invoked directly, without being wrapped by or given to a tuner, it will give an error.

The tuning range `ParamSet` that is constructed from the `TuneToken` values in a `ParamSet`'s `$values` slot can be accessed through the `ParamSet$search_space()` method. This is done automatically by tuners if no tuning range is given, but it is also possible to access the `$search_space()` method, modify it further, and give the modified `ParamSet` to a tuning function (or do anything else with it, nobody is judging you).

A `TuneToken` represents the range over which the parameter whose `$values` slot it occupies should be tuned over. It can be constructed via the `to_tune()` function in one of several ways:

- `to_tune()`: Indicates a parameter should be tuned over its entire range. Only applies to finite parameters (i.e. discrete or bounded numeric parameters)
- `to_tune(lower, upper, logscale)`: Indicates a numeric parameter should be tuned in the inclusive interval spanning `lower` to `upper`, possibly on a log scale if `logscale` is se to `TRUE`. All parameters are optional, and the parameter's own lower / upper bounds are used without log scale, by default. Depending on the parameter, integer (if it is a `p_int()`) or real values (if it is a `p_dbl()`) are used.
  `lower`, `upper`, and `logscale` can be given by position, except when only one of them is given, in which case it must be named to disambiguate from the following cases.
  When `logscale` is `TRUE`, then a `trafo` is generated automatically that transforms to the given bounds. The bounds are log()'d pre-trafo (see examples). See the `logscale` argument of `Domain` functions for more info.
  Note that "logscale" is *not* inherited from the `Domain` that the `TuneToken` belongs to! Defining a parameter with `p_dbl(... logscale = TRUE)` will *not* automatically give the `to_tune()` assigned to it log-scale.
- `to_tune(levels)`: Indicates a parameter should be tuned through the given discrete values. `levels` can be any named or unnamed atomic vector or list (although in the unnamed case it must be possible to construct a corresponding `character` vector with distinct values using `as.character`).
- `to_tune(<Domain>)`: The given `Domain` object (constructed e.g. with `p_int()` or `p_fct()`) indicates the range which should be tuned over. The supplied `trafo` function is used for parameter transformation.
- `to_tune(<ParamSet>)`: The given `ParamSet` is used to tune over a single dimension. This is useful for cases where a single evaluation-time parameter value (e.g. `p_uty()`) is constructed from multiple tuner-visible parameters (which may not be `p_uty()`). If not one-dimensional, the supplied `ParamSet` should always contain a `$extra_trafo` function, which must then always return a `list` with a single entry.

The TuneToken object's internals are subject to change and should not be relied upon. TuneToken objects should only be constructed via to_tune(), and should only be used by giving them to $values of a ParamSet.

Usage

```
to_tune(..., internal = !is.null(aggr), aggr = NULL)
```

Arguments

...	if given, restricts the range to be tuning over, as described above.
internal	(logical(1)) Whether to create an InternalTuneToken. This is only available for parameters tagged with "internal_tuning".
aggr	(function) Function with one argument, which is a list of parameter values and returns a single aggregated value (e.g. the mean). This specifies how multiple parameter values are aggregated to form a single value in the context of internal tuning. If none specified, the default aggregation function of the parameter will be used.

Value

A TuneToken object.

See Also

Other ParamSet construction helpers: Domain(), ps()

Examples

```
params = ps(
  int = p_int(0, 10),
  int_unbounded = p_int(),
  dbl = p_dbl(0, 10),
  dbl_unbounded = p_dbl(),
  dbl_bounded_below = p_dbl(lower = 1),
  fct = p_fct(c("a", "b", "c")),
  uty1 = p_uty(),
  uty2 = p_uty(),
  uty3 = p_uty(),
  uty4 = p_uty(),
  uty5 = p_uty()
)

params$values = list(

  # tune over entire range of `int`, 0..10:
  int = to_tune(),

  # tune over 2..7:
  int_unbounded = to_tune(2, 7),
```

```
  # tune on a log scale in range 1..10;
  # recognize upper bound of 10 automatically, but restrict lower bound to 1:
  dbl = to_tune(lower = 1, logscale = TRUE),
  ## This is equivalent to the following:
  # dbl = to_tune(p_dbl(log(1), log(10), trafo = exp)),

  # nothing keeps us from tuning a dbl over integer values
  dbl_unbounded = to_tune(p_int(1, 10)),

  # tune over values "a" and "b" only
  fct = to_tune(c("a", "b")),

  # tune over integers 2..8.
  # ParamUty needs type information in form of p_xxx() in to_tune.
  uty1 = to_tune(p_int(2, 8)),

  # tune uty2 like a factor, trying 1, 10, and 100:
  uty2 = to_tune(c(1, 10, 100)),

  # tune uty3 like a factor. The factor levels are the names of the list
  # ("exp", "square"), but the trafo will generate the values from the list.
  # This way you can tune an objective that has function-valued inputs.
  uty3 = to_tune(list(exp = exp, square = function(x) x^2)),

  # tune through multiple parameters. When doing this, the ParamSet in tune()
  # must have the trafo that generates a list with one element and the right
  # name:
  uty4 = to_tune(ps(
    base = p_dbl(0, 1),
    exp = p_int(0, 3),
    .extra_trafo = function(x, param_set) {
      list(uty4 = x$base ^ x$exp)
    }
  )),

  # not all values need to be tuned!
  uty5 = 100
)

print(params$values)

print(params$search_space())

# Change `$values` directly and generate new `$search_space()` to play around
params$values$uty3 = 8
params$values$uty2 = to_tune(c(2, 4, 8))

print(params$search_space())

# Notice how `logscale` applies `log()` to lower and upper bound pre-trafo:
params = ps(x = p_dbl())
```

```
params$values$x = to_tune(1, 100, logscale = TRUE)

print(params$search_space())

grid = generate_design_grid(params$search_space(), 3)

# The grid is equidistant within log-bounds pre-trafo:
print(grid)

# But the values are on a log scale scale with desired bounds after trafo:
print(grid$transpose())
```

Index

∗ **ParamSet construction helpers**
  Domain, 7
  ps, 30
  to_tune, 45
∗ **Sampler**
  Sampler, 35
  Sampler1D, 37
  Sampler1DCateg, 38
  Sampler1DNormal, 39
  Sampler1DRfun, 40
  Sampler1DUnif, 41
  SamplerHierarchical, 42
  SamplerJointIndep, 43
  SamplerUnif, 44
∗ **generate_design**
  generate_design_grid, 13
  generate_design_lhs, 14
  generate_design_random, 15
  generate_design_sobol, 15

assert_param_set, 3

CondAnyOf, *10*
CondAnyOf *(*condition_test*)*, 4
CondEqual, *10*
CondEqual *(*condition_test*)*, 4
Condition, *18*, *27, 28*
Condition *(*condition_test*)*, 4
condition_as_string *(*condition_test*)*, 4
condition_as_string(), *4*
condition_test, 4

data.table, *23*
data.table::data.table, *24, 25*
data.table::data.table(), *6*, *17, 18*, *28*
default_values, 5
Design, 6, *13–16*, *36*
Domain, *4*, 7, *13*, *16–18*, *20, 21*, *25, 26*, *30, 31*, *34*, *45, 46*

generate_design_grid, 13, *14–16*
generate_design_grid(), *6*
generate_design_lhs, *13*, 14, *15, 16*
generate_design_random, *13, 14*, 15, *16*
generate_design_sobol, *13–15*, 15

is_nodefault *(*NO_DEF*)*, 16

lhs::maximinLHS(), *14*

NO_DEF, *9*, 16
NoDefault *(*NO_DEF*)*, 16

p_dbl *(*Domain*)*, 7
p_dbl(), *17–19*, *39*, *45*
p_fct *(*Domain*)*, 7
p_fct(), *19*, *38*, *45*
p_int *(*Domain*)*, 7
p_int(), *18, 19*, *45*
p_lgl *(*Domain*)*, 7
p_lgl(), *17*, *19*, *38*
p_uty *(*Domain*)*, 7
p_uty(), *45*
paradox *(*paradox-package*)*, 3
paradox-package, 3
paradox::ParamSet, *28*
paradox::Sampler, *37–44*
paradox::Sampler1D, *38–41*
paradox::Sampler1DRfun, *39*
paradox::SamplerHierarchical, *44*
ParamSet, *4*, *6, 7*, *9, 10*, *13–16*, 17, *18*, *22*, *26*, *28–42*, *44–46*
ParamSetCollection, *26*, 28, *32*, *34*
ps, *11*, 30, *46*
ps(), *7*, *17*
ps_replicate, 32
ps_union, 34
psc, 32

R6, *6*, *20*, *29*, *36–44*

Sampler, 35, *37–44*

Sampler1D, *35, 36*, 37, *38–44*
Sampler1DCateg, *36, 37*, 38, *39–44*
Sampler1DNormal, *36–38*, 39, *40–44*
Sampler1DRfun, *36–39*, 40, *41–44*
Sampler1DUnif, *36–40*, 41, *42–44*
SamplerHierarchical, *35–41*, 42, *43, 44*
SamplerJointIndep, *35–42*, 43, *44*
SamplerUnif, *15*, *36–43*, 44
spacefillr::generate_sobol_set, *15*

to_tune, *11*, *31*, 45
to_tune(), *7*, *26*, *31*
TuneToken, *10*, *17*, *26*
TuneToken *(*to_tune*)*, 45

vname, *24, 25*