# Package ‘rgp’

August 8, 2014

**Version** 0.4-1

**Title** R genetic programming framework

**Description** RGP is a simple modular Genetic Programming (GP) system build in pure R. In addition to general GP tasks, the system supports Symbolic Regression by GP through the familiar R model formula interface. GP individuals are represented as R expressions, an (optional) type system enables domain-specific function sets containing functions of diverse domain- and range types. A basic set of genetic operators for variation (mutation and crossover) and selection is provided.

**Author** Oliver Flasch, Olaf Mersmann, Thomas Bartz-Beielstein, Joerg Stork,Martin Zaefferer

**Maintainer** Oliver Flasch <`oliver.flasch@fh-koeln.de`>

**License** GPL-2

**URL** `http://rsymbolic.org/projects/show/rgp`

**LazyData** yes

**Depends** R (>= 3.0.0), utils

**Imports** emoa (>= 0.5-0)

**Suggests** igraph (>= 0.5.5), rrules (>= 0.1-0), rgpui (>= 0.1-0),snowfall (>= 1.84-4)

**Date**

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2014-08-08 14:05:23

## R topics documented:

---

`rgp-package` *The RGP package*

---

## Description

RGP is a simple yet flexible modular Genetic Programming system for the R environment. The system implements classical untyped tree-based genetic programming as well as more advanced variants including, for example, strongly typed genetic programming and Pareto genetic programming.

## Author(s)

Oliver Flasch <`oliver.flasch@fh-koeln.de`>, Olaf Mersmann <`olafm@statistik.tu-dortmund.de`>,
Thomas Bartz-Beielstein <`thomas.bartz-beielstein@fh-koeln.de`>, Martin Zaefferer <`martin.zaefferer@fh-koeln.`
Joerg Stork <`joerg.stork@fh-koeln.de`>

---

`arithmeticFunctionSet` *Default function- and constant factory sets for Genetic Programming*

---

## Description

`arithmeticFunctionSet` is an untyped function set containing the functions "+", "-", "*", and "/". `expLogFunctionSet` is an untyped function set containing the functions "sqrt", "exp", and "ln". `trigonometricFunctionSet` is an untyped function set containing the functions "sin", "cos", and "tan". `mathFunctionSet` is an untyped function set containing all of the above functions.

## Usage

```
arithmeticFunctionSet

expLogFunctionSet
```

### Format

```
NULL
```

### Details

`numericConstantSet` is an untyped constant factory set containing a single constant factory that creates numeric constants via calls to `runif(1, -1, 1)`.

Note that these objects are initialized in the RGP package's `.onAttach` function.

---

## `arity` *Determine the number of arguments of a function*

---

### Description

Tries to determine the number of arguments of function.

### Usage

```
arity(f)
```

### Arguments

| | |
|---|---|
| `f` | The function to determine the arity for. |

### Value

The arity of the function `f`.

---

## `arity.primitive` *Determine the number of arguments of a primitive function*

---

### Description

Tries to determine the number of arguments of a primitive R function by lookup in a builtin table.

### Usage

```
arity.primitive(f)
```

### Arguments

| | |
|---|---|
| `f` | The primitive to determine the arity for. |

### Value

The arity of the primitive `f`.

breed *Breeding of GP individuals*

## Description

Breeds GP individuals by repeated application of an individual factory function. `individualFactory`. The `breedingFitness` must be a function of domain logical (a single boolean value) or numeric (a single real number). In case of a boolean breeding function, candidate individuals are created via the `individualFactory` function and tested by the `breedingFitness` predicate until the `breedingFitness` predicate is `TRUE` or `breedingTries` tries were done, in which case the last individual created and tested is returned. In case of a numerical breeding function, `breedingTries` individuals are created and evaluated by the `breedingFitness` function. The individual with the minimal breeding fitness is returned.

## Usage

```
breed(individualFactory, breedingFitness, breedingTries, warnOnFailure = TRUE,
  stopOnFailure = FALSE)
```

## Arguments

| | |
|---|---|
| `individualFactory` | A function of no parameters that returns a single GP individual. |
| `breedingFitness` | Either a function that takes a GP individual as its only parameter and returns a single logical value or a function that takes a GP individual as its only parameter and returns a single real value. |
| `breedingTries` | The number of breeding steps to perform. In case of a boolean `breedingFitness` function, the actual number of breeding steps performed may be lower then this number (see the details). |
| `warnOnFailure` | Whether to issue a warning when a boolean `breedingFitness` predicate was not fulfilled after `breedingTries` tries. |
| `stopOnFailure` | Whether to stop with an error message when a boolean `breedingFitness` predicate was not fulfilled after `breedingTries` tries. |

## Value

The GP individual that was bred.

## `buildingBlock` *Support for GP buidling blocks*

### Description

Building blocks are a means for protecting expression subtrees from modification through variation operators. Often, certain functional units, represented as expression subtrees in GP individuals, should stay intact during evolutionary search. Building blocks at the leafs of expressions can be introduced by adding them to the input variable set. Support for building blocks is planned for a future release of RGP.

### Usage

```
buildingBlock(expr, hardness = 1)

buildingBlockq(expr, hardness = 1)
```

### Arguments

| | |
|---|---|
| `expr` | The expresion to transform to a building block. |
| `hardness` | The strength of the protection against varition inside the building block. Must be a numeric in the interval [0.0, 1.0]. A `hardness` of `1.0` (the default) means that the building block will never be subject to variation. |

### Details

`buildingBlock` transforms an R expression to a building block to be used as an element of the input variable (or function) set. The parameter `hardness` (a numerical value in the interval [0.0 , 1.0]) determines the protection strength against variation inside the building building block. When `hardness` is set to `1.0` (the default), the building block will never be subject to variaton through mutation or crossover. `buildingBlockq` is equivaltent to `buildingBlock`, but quotes it's argument `expr` first.

### Value

A building block.

## `buildingBlockTag` *Building block tags*

### Description

To implement buidling blocks, i.e. subexpression protected from variation, expression nodes may be tagged with `buildingBlockTags`. TODO

### Usage

```
buildingBlockTag(x)

buildingBlockTag(x) <- value

hasBuildingBlockTag(x)
```

### Arguments

| | |
|---|---|
| `x` | An expression node. |
| `value` | The value of the building block tag. Must be a numerical in the interval [0.0 1.0]. |

---

## `commonSubexpressions` *Similarity and Distance Measures for R Functions and Expressions*

---

### Description

These functions implement several similarity and distance measures for R functions (i.e. their body expressions). TODO check and document measure-theoretic properties of each measure defined here TODO these distance measures are metrics, some of them are norm-induced metrics `commonSubexpressions` returns the set of common subexpressions of `expr1` and `expr2`. This is not a metric by itself, but can be used to implement several subtree-based similarity metrics. of `expr1` and `expr2`. `sizeWeightedNumberOfcommonSubexpressions` returns the number of common subexpressions of `expr1` and `expr2`, weighting the size of each common subexpression. Note that for every expression *e*, `sizeWeightedNumberOfcommonSubexpressions(` *e* `,` *e* `) == exprVisitationLength(` *e* `)`. `normalizedNumberOfCommonSubexpressions` returns the ratio of the number of common subexpressions of `expr1` and `expr2` in relation to the number of subexpression in the larger expression of `expr1` and `expr2`. `normalizedSizeWeightedNumberOfcommonSubexpressions` returns the ratio of the size-weighted number of common subexpressions of `expr1` and `expr2` in relation to the visitation length of the larger expression of `expr1` and `expr2`. `NCSdist` and `SNCSdist` are distance metrics derived from `normalizedNumberOfCommonSubexpressions` and `normalizedSizeWeightedNumberOfCommon` respectively. `differingSubexpressions`, and codenumberOfDifferingSubexpressions are duals of the functions described above, based on counting the number of differing subexpressions of `expr1` and `expr2`. The possible functions "normalizedNumberOfDifferingSubexpressions" and "normalizedSizeWeightedNumberOfDifferingSubexpressions" where ommited because they are always equal to `NCSdist` and `SNCSdist` by definition. `trivialMetric` The "trivial" metric M(a, b) that is 0 iff a == b, 1 otherwise. `normInducedTreeDistance` Uses a norm on expression trees and a metric on tree node labels to induce a metric M on expression trees A and B: If both A and B are empty (represented as `NULL`), M(A, B) := 0. If exactly one of A or B is empty, M(A, B) := "the norm applied to the non-empty tree". If neither A or B is empty, the difference of their root node labels (as measured by `labelDistance`) is added to the sum of the differences of the children. The children lists are padded with empty trees to equalize their sizes. The summation operator can be changed via `distanceFoldOperator`. `normInducedFunctionDistance` Is wrapper that applies `normInducedTreeDistance` to the bodies of the given functions.

## Usage

```
commonSubexpressions(expr1, expr2)

numberOfCommonSubexpressions(expr1, expr2)

normalizedNumberOfCommonSubexpressions(expr1, expr2)

NCSdist(expr1, expr2)

sizeWeightedNumberOfCommonSubexpressions(expr1, expr2)

normalizedSizeWeightedNumberOfCommonSubexpressions(expr1, expr2)

SNCSdist(expr1, expr2)

differingSubexpressions(expr1, expr2)

numberOfDifferingSubexpressions(expr1, expr2)

sizeWeightedNumberOfDifferingSubexpressions(expr1, expr2)

trivialMetric(a, b)

normInducedTreeDistance(norm, labelDistance = trivialMetric,
  distanceFoldOperator = NULL)

normInducedFunctionDistance(norm, labelDistance = trivialMetric,
  distanceFoldOperator = NULL)
```

## Arguments

| | |
|---|---|
| `expr1` | An R expression. |
| `expr2` | An R expression. |
| `a` | An R object. |
| `b` | An R object. |
| `norm` | A norm to derive a tree distance metric from. |
| `labelDistance` | A metric for measuring distances of tree node labels, i.e. function names or constants. |
| `distanceFoldOperator` | The operator used by `normInducedTreeDistance` to combine the measures subtree distances, defaults to '+'. |

`crossover` *Random crossover (recombination) of functions and expressions*

## Description

Replace a random subtree of `func1` (`expr1`) with a random subtree of `func2` (`expr2`) and return the resulting function (expression), i.e. the modified `func1` (`expr1`). `crossoverexpr` handles crossover of expressions instead of functions. `crossoverexprFast` is a fast (i.e. implemented in efficient C code) albeit less flexible variant of `crossoverexpr`. `crossoverTyped` and `crossoverexprTyped` only exchage replace subtress if the sTypes of their root nodes match. `crossoverTwoPoint` is a variant of `crossover` that swaps subtrees at uniform randomly selected points and returns both children. `crossoverexprTwoPoint` works analogously for expressions.

## Usage

```
crossover(func1, func2, crossoverprob = 0.1,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

crossoverexpr(expr1, expr2, crossoverprob)

crossoverexprFast(expr1, expr2)

crossoverexprTwoPoint(expr1, expr2)

crossoverTyped(func1, func2, crossoverprob = 0.1,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

crossoverexprTyped(expr1, expr2, crossoverprob)
```

## Arguments

| | |
|---|---|
| `expr1` | The first parent R expression. |
| `func1` | The first parent R function. |
| `expr2` | The second parent R expression. |
| `func2` | The second parent R function. |
| `crossoverprob` | The probability of crossover at each node of the first parent function (expression). |
| `breedingFitness` | A breeding function. See the documentation for `geneticProgramming` for details. |
| `breedingTries` | The number of breeding steps. |

## Details

All RGP recombination operators operating on functions have the S3 class `c("recombinationOperator", "function")`.

## Value

The child function (expression) or functions (expressions).

---

`customDist` *A* `dist` *function that supports custom metrics*

---

## Description

This function computes and returns the distance matrix computed by using the given metric to compute the distances between the rows of a data list or vector. Note that in contrast to `dist`, `x` has to be a vector and the the distance `metric` is an arbitrary function that must be symmetric and definite.

## Usage

```
customDist(x, metric, diag = FALSE, upper = FALSE)
```

## Arguments

| | |
|---|---|
| `x` | A vector or list of objects. |
| `metric` | A metric, i.e. a function of two arguments that returns a numeric. Note that a metric must be definite and symmetric, otherwise the results will be undefined. |
| `diag` | `TRUE` iff the diagonal of the distance matrix should be printed by `print.dist`. |
| `upper` | `TRUE` iff the upper triangle of the distance matrix should be printed by `print.dist`. |

## Value

A distance matrix.

## See Also

`dist`

---

`dataDrivenGeneticProgramming`
*Data-driven untyped standard genetic programming*

---

## Description

Perform an untyped genetic programming using a fitness function that depends on a R data frame. Typical applications are data mining tasks such as symbolic regression or classification. The task is specified as a `formula` and a fitness function factory. Only simple formulas without interactions are supported. The result of the data-driven GP run is a model structure containing the formulas and an untyped GP population. This function is primarily an intermediate for extensions. End-users will probably use more specialized GP tools such as `symbolicRegression`.

## Usage

```
dataDrivenGeneticProgramming(formula, data, fitnessFunctionFactory,
  fitnessFunctionFactoryParameters = list(),
  stopCondition = makeTimeStopCondition(5), population = NULL,
  populationSize = 100, eliteSize = ceiling(0.1 * populationSize),
  elite = list(), extinctionPrevention = FALSE, archive = FALSE,
  functionSet = mathFunctionSet, constantSet = numericConstantSet,
  crossoverFunction = NULL, mutationFunction = NULL,
  restartCondition = makeEmptyRestartCondition(),
  restartStrategy = makeLocalRestartStrategy(),
  searchHeuristic = makeAgeFitnessComplexityParetoGpSearchHeuristic(),
  breedingFitness = function(individual) TRUE, breedingTries = 50,
  progressMonitor = NULL, verbose = TRUE)
```

## Arguments

| | |
|---|---|
| `formula` | A `formula` describing the task. Only simple formulas of the form `response ~ variable1 + ... + vari` are supported at this point in time. |
| `data` | A `data.frame` containing training data for the GP run. The variables in `formula` must match column names in this data frame. |
| `fitnessFunctionFactory` | A function that accepts two parameters, a codeformula, data (given as a model frame) and the additional parameters given in `fitnessFunctionFactoryParameters` and returns a fitness function. |
| `fitnessFunctionFactoryParameters` | Additional parameters to pass to the `fitnessFunctionFactory`. |
| `stopCondition` | The stop condition for the evolution main loop. See `makeStepsStopCondition` for details. |
| `population` | The GP population to start the run with. If this parameter is missing, a new GP population of size `populationSize` is created through random growth. |
| `populationSize` | The number of individuals if a population is to be created. |
| `eliteSize` | The number of elite individuals to keep. Defaults to `ceiling(0.1 * populationSize)`. |
| `elite` | The elite list, must be alist of individuals sorted in ascending order by their first fitness component. |
| `extinctionPrevention` | When set to `TRUE`, the initialization and selection steps will try to prevent duplicate individuals from occurring in the population. Defaults to `FALSE`, as this operation might be expensive with larger population sizes. |
| `archive` | If set to `TRUE`, all GP individuals evaluated are stored in an archive list `archiveList` that is returned as part of the result of this function. |
| `functionSet` | The function set. |
| `constantSet` | The set of constant factory functions. |
| `crossoverFunction` | The crossover function. |

| | |
|---|---|
| `mutationFunction` | The mutation function. |
| `restartCondition` | The restart condition for the evolution main loop. See makeEmptyRestartCondition for details. |
| `restartStrategy` | The strategy for doing restarts. See makeLocalRestartStrategy for details. |
| `searchHeuristic` | The search-heuristic (i.e. optimization algorithm) to use in the search of solutions. See the documentation for `searchHeuristics` for available algorithms. |
| `breedingFitness` | A "breeding" function. This function is applied after every stochastic operation *Op* that creates or modifies an individal (typically, *Op* is a initialization, mutation, or crossover operation). If the breeding function returns `TRUE` on the given individual, *Op* is considered a success. If the breeding function returns `FALSE`, *Op* is retried a maximum of `breedingTries` times. If this maximum number of retries is exceeded, the result of the last try is considered as the result of *Op*. In the case the breeding function returns a numeric value, the breeding is repeated `breedingTries` times and the individual with the lowest breeding fitness is considered the result of *Op*. |
| `breedingTries` | In case of a boolean `breedingFitness` function, the maximum number of retries. In case of a numerical `breedingFitness` function, the number of breeding steps. Also see the documentation for the `breedingFitness` parameter. Defaults to `50`. |
| `progressMonitor` | A function of signature `function(population, fitnessfunction, stepNumber, evaluationNumber` to be called with each evolution step. |
| `verbose` | Whether to print progress messages. |

## Value

A model structure that contains the formula and an untyped GP population.

## See Also

`geneticProgramming`

---

`do.call.ignore.unused.arguments`

*A variant of do.call that ignores unused arguments*

---

## Description

A variant of do.call that ignores unused arguments

### Usage

```
do.call.ignore.unused.arguments(what, args, quote = FALSE,
  envir = parent.frame())
```

### Arguments

| | |
|---|---|
| `what` | What to call (either a function or a character vector naming a function in `envir`. |
| `args` | The args for the call, these may include arguments not used by `what`. |
| `quote` | Whether to quote the arguments. |
| `envir` | The environment within which to evaluate the call. |

### Value

The result of the call.

## `embedDataFrame` *Embed columns in a data frame*

### Description

Embeds the columns named `cols` in the data frame `x` into a space of dimension `dimension`.

### Usage

```
embedDataFrame(x, cols = NULL, dimension = 1)
```

### Arguments

| | |
|---|---|
| `x` | The data frame containing the columns to embed. |
| `cols` | A vector a list of the names of the columns to embed. |
| `dimension` | The additional dimensions to generate when embedding. |

### Value

The data frame, augmented with embedded columns, shortended by `dimension` rows.

`exprChildrenOrEmptyList` *Return the Children of an Expression or the Empty List if there are None*

## Description

Internal tool function that returns the children expressions of an R expression or the empty list if there are no children, i.e. if the expression is atomic or `NULL`. If the expression is a "function" expression, i.e. an expression that would evaluate to a function, `exprChildrenOrEmptyList` will return the function body expression as the only child.

## Usage

```
exprChildrenOrEmptyList(expr)
```

## Arguments

| | |
|---|---|
| `expr` | The expression to return the children for. |

## Value

The expression's children as a list, or the empty list if there are none.

`exprDepth` *Complexity measures for R functions and expressions*

## Description

`exprDepth` returns the depth of the tree representation ("exression tree") of an R expression. `funcDepth` returns the tree depth of the body expression of an R function. `exprSize` returns the number of nodes in the tree of an R expression. `exprLeaves` returns the number of leave nodes in the tree of an R expression. `exprCount` returns the number of tree nodes in an R expression matching a given predicate. `funcSize` returns the number of nodes in the body expression tree of an R function. `funcLeaves` returns the number of leave nodes in the body expression tree of an R function. `funcCount` returns the number of nodes in an R function body expression matching a given predicate. `exprVisitationLength` returns the visitation length of the tree of an R expression. The visitation length is the total number of nodes in all possible subtrees of a tree. `funcVisitationLength` returns the visitation length of the body expression tree of an R function. `fastExprVisitationLength` and `fastFuncVisitationLength` are variants written in optimized C code. The visitation length can be interpreted as the size of the expression obtained by substituting all inner functions by their function bodies (see "Crossover Bias in Genetic Programming", Maarten Keijzer and James Foster).

### Usage

```
exprDepth(expr)

funcDepth(func)

exprSize(expr)

exprLeaves(expr)

exprCount(expr, predicate = function(node) TRUE)

funcSize(func)

funcLeaves(func)

funcCount(func, predicate = function(node) TRUE)

exprVisitationLength(expr, intermediateResults = FALSE)

fastExprVisitationLength(expr, intermediateResults = FALSE)

funcVisitationLength(func, intermediateResults = FALSE)

fastFuncVisitationLength(func, intermediateResults = FALSE)
```

### Arguments

| | |
|---|---|
| `expr` | An R expression. |
| `func` | An R function. |
| `predicate` | An R predicate (function with range type `logical`). |
| `intermediateResults` | Whether to return complexity measures for all subtrees also. |

## `exprLabel` *Return the "label" at the Root Node of an Expression Tree*

### Description

Internal tool function that returns the function name if `expr` is a call, or otherwise just `expr` itself.

### Usage

```
exprLabel(expr)
```

### Arguments

| | |
|---|---|
| `expr` | The expression to return the root label for. |

## Value

The expression's root label.

---

`exprShapesOfDepth` *Upper bounds for expression tree search space sizes*

---

## Description

These functions return the number of structurally different expressions or expression shapes of a given depth or size that can be build from a fixed function- and input-variable set. Here, "expression shape" means the shape of an expression tree, not taking any node labels into account. `exprShapesOfDepth` returns the number of structurally different expression shapes of a depth exactly equal to `n`. `exprShapesOfMaxDepth` returns the number of structurally different expression shapes of a depth less or equal to `n`. `exprsOfDepth` returns the number of structurally different expressions of a depth exactly equal to `n`. Note that constants are handled by conceptually substiting them with a fresh input variable. `exprShapesOfMaxDepth` returns the number of structurally different expressions of a depth less or equal to `n`. Note that constants are handled by conceptually substiting them with a fresh input variable. `exprShapesOfSize`, `exprShapesOfMaxSize`, `exprsOfSize`, `exprsOfMaxSize` are equivalents that regard expression tree size (number of nodes) instead of expression tree depth.

## Usage

```
exprShapesOfDepth(funcset, n)

exprShapesOfMaxDepth(funcset, n)

exprsOfDepth(funcset, inset, n)

exprsOfMaxDepth(funcset, inset, n)

exprShapesOfSize(funcset, n)

exprShapesOfMaxSize(funcset, n)

exprsOfSize(funcset, inset, n)

exprsOfMaxSize(funcset, inset, n)
```

## Arguments

| | |
|---|---|
| `funcset` | The function set. |
| `inset` | The set of input variables. |
| `n` | The fixed size or depth. |

## `exprToPlotmathExpr` *Convert any expression to an expression that is plottable by plotmath*

### Description

Tries to convert a GP-generated expression `expr` to an expression plottable by `plotmath` by replacing GP variants of arithmetic operators by their standard counterparts.

### Usage

```
exprToPlotmathExpr(expr)
```

### Arguments

| | |
|---|---|
| `expr` | The GP-generated expression to convert. |

### Value

An expression plottable by `plotmath`.

## `extractAttributes` *Extract a given attribute of all objects in a list and tag that list with the list of extracted attributes*

### Description

Extract a given attribute of all objects in a list and tag that list with the list of extracted attributes

### Usage

```
extractAttributes(x, extractAttribute, tagAttribute = extractAttribute,
  default = NULL)
```

### Arguments

| | |
|---|---|
| `x` | A list with objects containing the attribute `attribute`. |
| `extractAttribute` | The attribute to extract from all objects in the list `x`. |
| `tagAttribute` | The name of the attribute for `x` holding the list of extracted attributes. |
| `default` | A default value to return if an object in `x` has no attribute `attribute`. |

### Value

The list `x`, tagged with a new attribute `tagAttribute`.

## `first` *Functions for Lisp-like list processing*

### Description

Simple wrapper functions that allow Lisp-like list processing in R: `first` to `fifth` return the first to fifth element of the list `x`. `rest` returns all but the first element of the list `x`. `is.empty` returns `TRUE` iff the list `x` is of length 0. `is.atom` returns `TRUE` iff the list `x` is of length 1. `is.composite` returns `TRUE` iff the list `x` is of length > 1. `contains` return `TRUE` iff the list `x` contains an element identical to `elt`.

### Usage

```
first(x)

rest(x)

second(x)

third(x)

fourth(x)

fifth(x)

is.empty(x)

is.atom(x)

is.composite(x)

contains(x, elt)
```

### Arguments

| | |
|---|---|
| `x` | A list or vector. |
| `elt` | An element of a list or vector. |

## `formatSeconds` *Format time and data values into human-readable character vectors*

### Description

These functions convert date and time values into human-readable character vectors. `formatSeconds` formats time values given as a numerical vector denoting seconds into human-readable character vectors, i.e. `formatSeconds(70)` results in the string `"1 minute, 10 seconds"`.

## Usage

```
formatSeconds(seconds, secondDecimals = 2)
```

## Arguments

| | |
|---|---|
| `seconds` | A numeric vector denoting seconds. |
| `secondDecimals` | The number of decimal places to show for seconds. Defaults to 2. |

## Value

A character vector containg a human-readable representation of the given date/time.

---

`functionSet` *Functions for defining the search space for Genetic Programming*

---

## Description

The GP search space is defined by a set of functions, a set of input variables, a set of constant constructor functions, and some rules how these functions, input variables, and constants may be combined to form valid symbolic expressions. The function set is simply given as a set of strings naming functions in the global environment. Input variables are also given as strings. Combination rules are implemented by a type system and defined by assigning sTypes to functions, input variables, and constant constructors.

## Usage

```
functionSet(..., list = NULL, parentEnvironmentLevel = 1)

inputVariableSet(..., list = NULL)

constantFactorySet(..., list = NULL)

pw(x, pw)

hasPw(x)

getPw(x, default = 1)

## S3 method for class 'functionSet'
c(..., recursive = FALSE)

## S3 method for class 'inputVariableSet'
c(..., recursive = FALSE)

## S3 method for class 'constantFactorySet'
c(..., recursive = FALSE)
```

## Arguments

| | |
|---|---|
| `...` | Names of functions or input variables given as strings. |
| `list` | Names of functions or input variables given as a list of strings. |
| `parentEnvironmentLevel` | Level of the parent environment used to resolve function names. |
| `recursive` | Ignored when concatenating function- or input variable sets. |
| `x` | An object (function name, input variable name, or constant factory) to tag with a probability `pw`. |
| `pw` | A probability weight. |
| `default` | A default probability weight to return iff no probability weight is associated with an object. |

## Details

Function sets and input variable sets are S3 classes containing the following fields: `$all` contains a list of all functions, or input variables, or constant factories. `$byRange` contains a table of all input variables, or functions, or constant factories, indexed by the string label of their sTypes for input variables, or by the string label of their range sTypes for functions, or by the string label of their range sTypes for constant factories. This field exists mainly for quickly finding a function, input variable, or constant factory that matches a given type.

Multiple function sets, or multiple input variable sets, or multiple constant factory sets can be combined using the `c` function. `functionSet` creates a function set. `inputVariableSet` creates an input variable set. `constantFactorySet` creates a constant factory set.

Probability weight for functions, input variables, and constants can be given by tagging constant names, input variables, and constant factory functions via the `pw` function (see the examples). The predicate `hasPw` can be used to check if an object `x` has an associated probability weight. The function `getPw` returns the probability weight associated with an object `x`, if available.

## Value

A function set or input variable set.

## Examples

```
# creating an untyped search space description...
functionSet("+", "-", "*", "/", "exp", "log", "sin", "cos", "tan")
inputVariableSet("x", "y")
constantFactorySet(function() runif(1, -1, 1))
# creating an untyped function set with probability weights...
functionSet(pw("+", 1.2), pw("-", 0.8), pw("*", 1.0), pw("/", 1.0))
```

## `functionVariablePresenceMap` *Variable Presence Maps*

### Description

Counts the number of input variables (formal arguments) present in the body of a individual function. Applied to a population of individuals, this information is useful to identify driving variables in a modelling task. `functionVariablePresenceMap` returns a (one row) variable presence map for a function, `populationVariablePresenceMap` returns a variable presence map for a population of RGP individuals (a list of R functions).

### Usage

```
functionVariablePresenceMap(f)

populationVariablePresenceMap(pop)
```

### Arguments

| | |
|---|---|
| `f` | A R function to return a variable presence map for. |
| `pop` | A RGP population to return a variable presence map for. |

### Value

A data frame with variables (formal parameters) in the columns, individuals (function) in the rows and variable counts in the cells.

## `funcToIgraph` *Visualization of functions and expressions as trees*

### Description

The following functions plot R expressions and functions as trees. The igraph package is required for most of these functions. `exprToGraph` transforms an R expression into a graph given as a character vector of vertices V and a even-sized numeric vector of edges E. Two elements i and i+1 in E encode a directed edge from V[i] to V[i+1]. `funcToIgraph` and `exprToIgraph` return an igraph graph object for an R function or an R expression.

### Usage

```
funcToIgraph(func)

exprToIgraph(expr)

exprToGraph(expr)
```

### Arguments

| | |
|---|---|
| `func` | An R function. |
| `expr` | An R expression. |

### Value

The result (see the details section).

### See Also

`funcToPlotmathExpr`

---

`funcToPlotmathExpr` *Convert a function to an expression plottable by plotmath*

---

### Description

Tries to convert a function `func` to an expression plottable by `plotmath` by replacing arithmetic operators and "standard" functions by plottable counterparts.

### Usage

```
funcToPlotmathExpr(func)
```

### Arguments

| | |
|---|---|
| `func` | The function to convert. |

### Value

An expression plottable by `plotmath`.

### See Also

`funcToIgraph`

`geneticProgramming` *Standard typed and untyped genetic programming*

## Description

Perform a standard genetic programming (GP) run. Use `geneticProgramming` for untyped genetic programming or `typedGeneticProgramming` for typed genetic programming runs. The required argument `fitnessFunction` must be supplied with an objective function that assigns a numerical fitness value to an R function. Fitness values are minimized, i.e. smaller values denote higher/better fitness. If a multi-objective `selectionFunction` is used, `fitnessFunction` return a numerical vector of fitness values. The result of the GP run is a GP result object containing a GP population of R functions. `summary.geneticProgrammingResult` can be used to create summary views of a GP result object. During the run, restarts are triggered by the `restartCondition`. When a restart is triggered, the restartStrategy is executed, which returns a new population to replace the current one as well as a list of elite individuals. These are added to the runs elite list, where fitter individuals replace individuals with lesser fittness. The runs elite list is always sorted by fitness in ascending order. Only the first component of a multi-criterial fitness counts in this sorting. After a GP run, the population is inserted into the elite list. The elite list is returned as part of the GP result object.

## Usage

```
geneticProgramming(fitnessFunction, stopCondition = makeTimeStopCondition(5),
  population = NULL, populationSize = 100, eliteSize = ceiling(0.1 *
  populationSize), elite = list(), functionSet = mathFunctionSet,
  inputVariables = inputVariableSet("x"), constantSet = numericConstantSet,
  crossoverFunction = crossover, mutationFunction = NULL,
  restartCondition = makeEmptyRestartCondition(),
  restartStrategy = makeLocalRestartStrategy(),
  searchHeuristic = makeAgeFitnessComplexityParetoGpSearchHeuristic(lambda =
  ceiling(0.5 * populationSize)), breedingFitness = function(individual) TRUE,
  breedingTries = 50, extinctionPrevention = FALSE, archive = FALSE,
  progressMonitor = NULL, verbose = TRUE)

typedGeneticProgramming(fitnessFunction, type,
  stopCondition = makeTimeStopCondition(5), population = NULL,
  populationSize = 100, eliteSize = ceiling(0.1 * populationSize),
  elite = list(), functionSet, inputVariables, constantSet,
  crossoverFunction = crossoverTyped, mutationFunction = NULL,
  restartCondition = makeEmptyRestartCondition(),
  restartStrategy = makeLocalRestartStrategy(populationType = type),
  searchHeuristic = makeAgeFitnessComplexityParetoGpSearchHeuristic(),
  breedingFitness = function(individual) TRUE, breedingTries = 50,
  extinctionPrevention = FALSE, archive = FALSE, progressMonitor = NULL,
  verbose = TRUE)
```

## Arguments

| | |
|---|---|
| `fitnessFunction` | In case of a single-objective selection function, `fitnessFunction` must be a single function that assigns a numerical fitness value to a GP individual represented as a R function. Smaller fitness values mean higher/better fitness. If a multi-objective selection function is used, `fitnessFunction` must return a numerical vector of fitness values. |
| `type` | The range type of the individual functions. This parameter only applies to `typedGeneticProgramming`. |
| `stopCondition` | The stop condition for the evolution main loop. See code `makeStepsStopCondition` for details. |
| `population` | The GP population to start the run with. If this parameter is missing, a new GP population of size `populationSize` is created through random growth. |
| `populationSize` | The number of individuals if a population is to be created. |
| `eliteSize` | The number of elite individuals to keep. Defaults to `ceiling(0.1 * populationSize)`. |
| `elite` | The elite list, must be alist of individuals sorted in ascending order by their first fitness component. |
| `functionSet` | The function set. |
| `inputVariables` | The input variable set. |
| `constantSet` | The set of constant factory functions. |
| `crossoverFunction` | The crossover function. |
| `mutationFunction` | The mutation function. |
| `restartCondition` | The restart condition for the evolution main loop. See makeEmptyRestartCondition for details. |
| `restartStrategy` | The strategy for doing restarts. See makeLocalRestartStrategy for details. |
| `searchHeuristic` | The search-heuristic (i.e. optimization algorithm) to use in the search of solutions. See the documentation for `searchHeuristics` for available algorithms. |
| `breedingFitness` | A "breeding" function. This function is applied after every stochastic operation *Op* that creates or modifies an individal (typically, *Op* is a initialization, mutation, or crossover operation). If the breeding function returns `TRUE` on the given individual, *Op* is considered a success. If the breeding function returns `FALSE`, *Op* is retried a maximum of `breedingTries` times. If this maximum number of retries is exceeded, the result of the last try is considered as the result of *Op*. In the case the breeding function returns a numeric value, the breeding is repeated `breedingTries` times and the individual with the lowest breeding fitness is considered the result of *Op*. |
| `breedingTries` | In case of a boolean `breedingFitness` function, the maximum number of retries. In case of a numerical `breedingFitness` function, the number of breeding steps. Also see the documentation for the `breedingFitness` parameter. Defaults to `50`. |

| | |
|---|---|
| `extinctionPrevention` | When set to `TRUE`, the initialization and selection steps will try to prevent duplicate individuals from occurring in the population. Defaults to `FALSE`, as this operation might be expensive with larger population sizes. |
| `archive` | If set to `TRUE`, all GP individuals evaluated are stored in an archive list `archiveList` that is returned as part of the result of this function. |
| `progressMonitor` | A function of signature `function(population, objectiveVectors, fitnessFunction, stepNumber` to be called with each evolution step. Seach heuristics may pass additional information via the `...` parameter. |
| `verbose` | Whether to print progress messages. |

### Value

A genetic programming result object that contains a GP population in the field `population`, as well as metadata describing the run parameters.

### See Also

`summary.geneticProgrammingResult`, `symbolicRegression`

---

## `gridDesign` *Create a regular grid design matrix*

---

### Description

Returns a n = length(points)**dimension times m = dimension matrix containing the coordinates of sample points from a hypervolume of the given dimension. Points are sampled in a grid defined by the vector points.

### Usage

```
gridDesign(dimension, points = seq(from = 0, to = 1, length.out = 10))
```

### Arguments

| | |
|---|---|
| `dimension` | The number of columns in the design matrix to create. |
| `points` | A vector of points to sample at in each dimension. |

### Value

The regular grid design matrix.

`inputVariablesOfIndividual` *Functions for analysing GP individuals*

## Description

`inputVariablesOfIndividual` returns a list of input variables in `inset` that are used by the GP individual `ind`.

## Usage

```
inputVariablesOfIndividual(ind, inset)
```

## Arguments

| | |
|---|---|
| `ind` | A GP individual, represented as a R function. |
| `inset` | A set of input variables. |

`insertionSort` *Sorting algorithms for vectors and lists*

## Description

These algorithms sort a list or vector by a given order relation (which defaults to <=). `insertionSort` is a stable O(n^2) sorting algorithm that is quite efficient for very small sets (less than around 20 elements). Use an O(n*log(n)) algorithm for larger sets.

## Usage

```
insertionSort(xs, orderRelation = NULL)
```

## Arguments

| | |
|---|---|
| `xs` | The vector or list to sort. |
| `orderRelation` | The orderRelation to sort `xs` by (defaults to \`<=\`). This relation by should reflexive, antisymetric, and transitive. |

## Value

The vector or list `xs` sorted by the order relation `orderRelation`.

## `integerToLogicals` *Tools for manipulating boolean functions*

### Description

`integerToBoolean` converts a scalar positive integer (or zero) to its binary representation as list of logicals. `booleanFunctionVector` returns the boolean vector of result values of `f`, given a boolean function `f`. `numberOfDifferentBits` given two lists of booleans of equal length, returns the number of differing bits. `makeBooleanFitnessFunction` given a boolean target function, returns a fitness function that returns the number of different places in the output of a given boolean function and the target function.

### Usage

```
integerToLogicals(i, width = floor(log(base = 2, i) + 1))

booleanFunctionAsList(f)

numberOfDifferentBits(a, b)

makeBooleanFitnessFunction(targetFunction)
```

### Arguments

| | |
|---|---|
| `i` | A scalar positive integer. |
| `width` | The with of the logical vector to return. |
| `f` | A boolean function. |
| `a` | A list of booleans. |
| `b` | A list of booleans. |
| `targetFunction` | A boolean function. |

### Value

The function result as described above.

## `inversePermutation` *Calculate the inverse of a permutation*

### Description

Returns the inverse of a permutation `x` given as an integer vector. This function is useful to turn a ranking into an ordering and back, for example.

### Usage

```
inversePermutation(x)
```

### Arguments

| | |
|---|---|
| `x` | The permutation to return the inverse for. |

### Value

The inverse of the permutation `x`.

### See Also

`rank`, `order`

---

`is.sType` *Check if an object is an sType*

---

### Description

Returns `TRUE` iff its argument is an sType.

### Usage

```
is.sType(x)
```

### Arguments

| | |
|---|---|
| `x` | The object to check. |

### Value

`TRUE` iff `x` is an sType.

## iterate — *Repeatedly apply a function*

### Description

Repeatedly apply a function `f` to an argument `arg`, additional arguments `...` are supplied unchanged in each call. E.g. `iterate(3, foo, 42.14, "bar")` is equivalent to `foo(foo(foo(42.14, "bar"), "bar"), "ba`

### Usage

```
iterate(n, f, arg, ...)
```

### Arguments

| | |
|---|---|
| `n` | The number of times to apply `f`, must be >= 0. If 0, `arg` is returned. |
| `f` | The function to apply. |
| `arg` | The argument to repeatedly apply `f` to. |
| `...` | Additional argument to pass to `f` at each application. |

### Value

The result of repeatedly applying `f`.

## joinElites — *Join elite lists*

### Description

Inserts a list of new individuals into an elite list, replacing the worst individuals in this list to make place, if needed.

### Usage

```
joinElites(individuals, elite, eliteSize, fitnessFunction)
```

### Arguments

| | |
|---|---|
| `individuals` | The list of individuals to insert. |
| `elite` | The list of elite individuals to insert `individuals` into. This list must be sorted by fitness in ascending order, i.e. lower fitnesses first. |
| `eliteSize` | The maximum size of the `elite`. |
| `fitnessFunction` | The fitness function. |

### Value

The `elite` with `individuals` inserted, sorted by fitness in ascending order, i.e. lower fitnesses first.

---

## `latinHypercubeDesign` *Create a latin hypercube design (LHD)*

---

### Description

Produces a LHD matrix with `dimension` columns and `size` rows.

### Usage

```
latinHypercubeDesign(dimension, size = max(11 * dimension, 1 + 3 * dimension +
  dimension * (dimension - 1)/2 + 1), lowerBounds = replicate(dimension, 0),
  upperBounds = replicate(dimension, 1), retries = 2 * dimension)
```

### Arguments

| | |
|---|---|
| `dimension` | Dimension of the problem (will be no. of columns of the result matrix). |
| `size` | Number of design points, defaults to `max(11 * dimension,1 + 3 * dimension + dimension * (dimer` |
| `lowerBounds` | Numeric vector of length `dimension` giving lower bounds for sampling, defaults to `c(0.0, ...)`. |
| `upperBounds` | Numeric vector of length `dimension` giving upper bounds for sampling, defaults to `c(1.0, ...)`. |
| `retries` | Number of retries, which is the number of trials to find a design with the lowest distance, default is `2 * dimension`. |

### Value

A LHD matrix.

---

## `mae` *Mean absolute error (MAE)*

---

### Description

Mean absolute error (MAE)

### Usage

```
mae(x, y)
```

## Arguments

| | |
|---|---|
| x | A numeric vector or list. |
| y | A numeric vector or list. |

## Value

The MAE between x and y.

---

makeAgeFitnessComplexityParetoGpSearchHeuristic
*Age Fitness Complexity Pareto GP Search Heuristic for RGP*

---

## Description

The search-heuristic, i.e. the concrete GP search algorithm, is a modular component of RGP. makeAgeFitnessComplexityParetoGpSearchHeuristic creates a RGP search-heuristic that implements a generational evolutionary multi objective optimization algorithm (EMOA) that selects on three criteria: Individual age, individual fitness, and individual complexity.

## Usage

```
makeAgeFitnessComplexityParetoGpSearchHeuristic(lambda = 50,
  crossoverProbability = 0.5, enableComplexityCriterion = TRUE,
  enableAgeCriterion = FALSE, ndsParentSelectionProbability = 0,
  ndsSelectionFunction = nds_cd_selection, complexityMeasure = function(ind,
  fitness) fastFuncVisitationLength(ind), ageMergeFunction = max,
  newIndividualsPerGeneration = if (enableAgeCriterion) 50 else 0,
  newIndividualsMaxDepth = 8, newIndividualFactory = makePopulation)
```

## Arguments

| | |
|---|---|
| lambda | The number of children to create in each generation (50 by default). |
| crossoverProbability | The crossover probability for search-heuristics that support this setting (i.e. TinyGP). Defaults to 0.5. |
| enableComplexityCriterion | Whether to enable the complexity criterion in multi-criterial search heuristics. |
| enableAgeCriterion | Whether to enable the age criterion in multi-criterial search heuristics. |
| ndsParentSelectionProbability | The probability to use non-dominated sorting to select parents for each generation. When set to 0.0, parents are selected by uniform random sampling without replacement every time. Defaults to 1.0. |
| ndsSelectionFunction | The function to use for non-dominated sorting in Pareto GP selection. Defaults to nds_cd_selection. |

`complexityMeasure`
: The complexity measure, a function of signature `function(ind, fitness)` returning a single numeric value.

`ageMergeFunction`
: The function used for merging ages of crossover children, defaults to `max`.

`newIndividualsPerGeneration`
: The number of new individuals per generation to insert into the population. Defaults to `50` if `enableAgeCriterion == TRUE` else to `0`.

`newIndividualsMaxDepth`
: The maximum depth of new individuals inserted into the population.

`newIndividualFactory`
: The factory function for creating new individuals. Defaults to `makePopulation`.

## Value

An RGP search heuristic.

---

`makeArchiveBasedParetoTournamentSearchHeuristic`
*Archive-based Pareto Tournament Search Heuristic for RGP*

---

## Description

The search-heuristic, i.e. the concrete GP search algorithm, is a modular component of RGP. `makeArchiveBasedParetoTournamentSearchHeuristic` creates a RGP search-heuristic that implements a archive-based Pareto tournament multi objective optimization algorithm (EMOA) that selects on three criteria: Individual fitness, individual complexity and individual age.

## Usage

```
makeArchiveBasedParetoTournamentSearchHeuristic(archiveSize = 50,
  popTournamentSize = 5, archiveTournamentSize = 3, crossoverRate = 0.95,
  enableComplexityCriterion = TRUE, complexityMeasure = function(ind,
  fitness) fastFuncVisitationLength(ind),
  ndsSelectionFunction = nds_cd_selection)
```

## Arguments

`archiveSize`
: The number of individuals in the archive, defaults to `50`.

`popTournamentSize`
: The size of the Pareto tournaments for selecting individuals for reproduction from the population.

`archiveTournamentSize`
: The size of the Pareto tournaments for selecting individuals for reproduction from the archive.

| | |
|---|---|
| `crossoverRate` | The probabilty to do crossover with an archive member instead of mutation of an archive member. |
| `enableComplexityCriterion` | Whether to enable the complexity criterion in multi-criterial search heuristics. |
| `complexityMeasure` | The complexity measure, a function of signature `function(ind, fitness)` returning a single numeric value. |
| `ndsSelectionFunction` | The function to use for non-dominated sorting in Pareto GP selection. Defaults to `nds_cd_selection`. |

## Value

An RGP search heuristic.

---

`makeClosure` *Create a new R closure given a function body expression and an argument list*

---

## Description

Creates a R closure (i.e. a function object) from a body expression and an argument list. The closure's environment will be the default environment.

## Usage

```
makeClosure(fbody, fargs, envir = globalenv())
```

## Arguments

| | |
|---|---|
| `fbody` | The function body, given as a R expression. |
| `fargs` | The formal arguments, given as a list or vector of strings. |
| `envir` | The new function closure's environment, defaults to `globalenv()`. |

## Value

A formal argument list, ready to be passed via `formals`.

## `makeCommaEvolutionStrategySearchHeuristic` *Comma Evolution Strategy Search Heuristic for RGP*

### Description

The search-heuristic, i.e. the concrete GP search algorithm, is a modular component of RGP. `makeCommaEvolutionStrategySearchHeuristic` creates a RGP search-heuristic that implements a (mu, lambda) Evolution Strategy. The lambda parameter is fixed to the population size. TODO description based on Luke09a

### Usage

```
makeCommaEvolutionStrategySearchHeuristic(mu = 1)
```

### Arguments

| | |
|---|---|
| `mu` | The number of surviving parents for the Evolution Strategy search-heuristic. Note that with `makeCommaEvolutionStrategySearchHeuristic`, lambda is fixed to the population size, i.e. `length(pop)`. |

### Value

An RGP search heuristic.

## `makeEmptyRestartCondition` *Evolution restart conditions*

### Description

Evolution restart conditions are predicates (functions that return a single logical value) of the signature `function(population, fitnessFunction, stepNumber, evaluationNumber,bestFitness, timeElapsed)`. They are used to decide when to restart a GP evolution run that might be stuck in a local optimum. Evolution restart conditions are objects of the same type and class as evolution stop conditions. They may be freely substituted for each other.

### Usage

```
makeEmptyRestartCondition()

makeStepLimitRestartCondition(stepLimit = 10)

makeFitnessStagnationRestartCondition(fitnessHistorySize = 100,
  testFrequency = 10, fitnessStandardDeviationLimit = 1e-06)

makeFitnessDistributionRestartCondition(testFrequency = 100,
  fitnessStandardDeviationLimit = 1e-06)
```

### Arguments

| | |
|---|---|
| `stepLimit` | The step limit for `makeStepLimitRestartCondition`. |
| `fitnessHistorySize` | The number of past best fitness values to look at when calculating the best fitness standard deviation for `makeFitnessStagnationRestartCondition`. |
| `testFrequency` | The frequency to test for the restart condition, in evolution steps. This parameter is mainly used with restart condititions that are expensive to calculate. |
| `fitnessStandardDeviationLimit` | The best fitness standard deviation limit for `makeFitnessStagnationRestartCondition`. |

### Details

`makeEmptyRestartCondition` creates a restart condition that is never fulfilled, i.e. restarts will never occur. `makeStepLimitRestartCondition` creates a restart condition that holds if the number if evolution steps is an integer multiple of a given step limit. restarts will never occur. `makeFitnessStagnationRestartCond` creates a restart strategy that holds if the standard deviation of a last `fitnessHistorySize` best fitness values falls below a given `fitnessStandardDeviationLimit`. `makeFitnessDistributionRestartCondition` creates a restart strategy that holds if the standard deviation of the fitness values of the individuals in the current population falls below a given `fitnessStandardDeviationLimit`.

---

## `makeFunctionFitnessFunction` *Create a fitness function from a reference function of one variable*

---

### Description

Creates a fitness function that calculates an error measure with respect to an arbitrary reference function of one variable on the sequence of fitness cases `seq(from, to, length = steps)`. When an `indsizelimit` is given, individuals exceeding this limit will receive a fitness of `Inf`.

### Usage

```
makeFunctionFitnessFunction(func, from = -1, to = 1, steps = 128,
  errorMeasure = rmse, indsizelimit = NA)
```

### Arguments

| | |
|---|---|
| `func` | The reference function. |
| `from` | The start of the sequence of fitness cases. |
| `to` | The end of the sequence of fitness cases. |
| `steps` | The number of steps in the sequence of fitness cases. |
| `errorMeasure` | A function to use as an error measure, defaults to RMSE. |
| `indsizelimit` | Individuals exceeding this size limit will get a fitness of `Inf`. |

### Value

A fitness function based on the reference function `func`.

`makeHierarchicalClusterFunction`

*Clustering Populations for Niching*

## Description

These functions create `clusterFunction`s for `multiNicheGeneticProgramming` and `multiNicheSymbolicRegression`. `makeHierarchicalClusterFunction` returns a clustering function that uses Ward's agglomerative hierarchical clustering algorithm `hclust`.

## Usage

```
makeHierarchicalClusterFunction(distanceMeasure = NULL, minNicheSize = 1)
```

## Arguments

| | |
|---|---|
| `distanceMeasure` | A distance measure, used for calculating distances between individuals in a population. |
| `minNicheSize` | The minimum number of individuals in each niche. |

## Value

A `clusterFunction` for clustering populations.

## See Also

`multiNicheGeneticProgramming`, `multiNicheSymbolicRegression`

`makeLocalRestartStrategy`

*Evolution restart strategies*

## Description

Evolution restart strategies are functions of the signature `function(fitnessFunction,population, populationSize, fu` that return a list of two obtjects: First, a population that replace the run's current population. Second, a list of elite individuals to keep.

## Usage

```
makeLocalRestartStrategy(populationType = NULL,
  extinctionPrevention = FALSE, breedingFitness = function(individual) TRUE,
  breedingTries = 50)
```

### Arguments

| | |
|---|---|
| `populationType` | The sType of the replacement individuals, defaults to `NULL` for creating untyped populations. |
| `extinctionPrevention` | Whether to surpress duplicate individuals in newly initialized populations. See `geneticProgramming` for details. |
| `breedingFitness` | A breeding function. See the documentation for `geneticProgramming` for details. |
| `breedingTries` | The number of breeding steps. |

### Details

`makeLocalRestartStrategy` creates a restart strategy that replaces all individuals with new individuals. The single best individual is returned as the elite. When using a multi-criterial fitness function, only the first component counts in the fitness sorting.

---

## makeNaryFunctionFitnessFunction

*Create a fitness function from a n-ary reference function*

---

### Description

Creates a fitness function that calculates an error measure with respect to an arbitrary n-ary reference function based sample points generated by a given `designFunction`. When an `indsizelimit` is given, individuals exceeding this limit will receive a fitness of `Inf`.

### Usage

```
makeNaryFunctionFitnessFunction(func, dim, designFunction = gridDesign,
  errorMeasure = rmse, indsizelimit = NA, ...)
```

### Arguments

| | |
|---|---|
| `func` | The reference function. Its single argument must be numeric vector of length `dim` and it must return a scalar numeric. |
| `dim` | The dimension of the reference function. |
| `designFunction` | A function to generate sample points. Its first argument must be `dim`. Defaults to `gridDesign`. |
| `errorMeasure` | A function to use as an error measure, defaults to RMSE. |
| `indsizelimit` | Individuals exceeding this size limit will get a fitness of `Inf`. |
| `...` | Additional arguments to the `designFunction`. |

### Value

A fitness function based on the reference function `func`.

### See Also

`latinHypercubeDesign`, `gridDesign`,

---

## makePopulation    *Classes for populations of individuals represented as functions*

---

### Description

`makePopulation` creates a population of untyped individuals, whereas `makeTypedPopulation` creates a population of typed individuals. `fastMakePopulation` is a faster variant of `makePopulation` with fewer options. `print.population` prints the population. `summary.population` returns a summary view of a population.

### Usage

```
makePopulation(size, funcset, inset, conset, maxfuncdepth = 8,
  constprob = 0.2, breedingFitness = function(individual) TRUE,
  breedingTries = 50, extinctionPrevention = FALSE, funcfactory = NULL)

fastMakePopulation(size, funcset, inset, maxfuncdepth, constMin, constMax)

makeTypedPopulation(size, type, funcset, inset, conset, maxfuncdepth = 8,
  constprob = 0.2, breedingFitness = function(individual) TRUE,
  breedingTries = 50, extinctionPrevention = FALSE, funcfactory = NULL)

## S3 method for class 'population'
print(x, ...)

## S3 method for class 'population'
summary(object, ...)
```

### Arguments

| | |
|---|---|
| `size` | The population size in number of individuals. |
| `type` | The (range) type of the individual functions to create. |
| `funcset` | The function set. |
| `inset` | The set of input variables. |
| `conset` | The set of constant factories. |
| `constMin` | For `fastMakePopulation`, the minimum constant to create. |
| `constMax` | For `fastMakePopulation`, the maximum constant to create. |
| `maxfuncdepth` | The maximum depth of the functions of the new population. |
| `constprob` | The probability of generating a constant in a step of growth, if no subtree is generated. If neither a subtree nor a constant is generated, a randomly chosen input variable will be generated. Defaults to `0.2`. |

| | |
|---|---|
| `breedingFitness` | A breeding function. See the documentation for `geneticProgramming` for details. |
| `breedingTries` | The number of breeding steps. |
| `extinctionPrevention` | When set to `TRUE`, initialization will try to prevent duplicate individuals from occurring in the population. Defaults to `FALSE`, as this operation might be expensive with larger population sizes. |
| `funcfactory` | A factory for creating the functions of the new population. Defaults to Koza's "ramped half-and-half" initialization strategy. |
| `x` | The population to print. |
| `object` | The population to summarize. |
| `...` | Additional parameters to the `print` or `summary` (passed on to their default implementation). |

## Value

A new population of functions.

---

`makeRegressionFitnessFunction`

*Create a fitness function for symbolic regression*

---

## Description

Creates a fitness function that calculates an error measure with respect to a given set of data variables. A simplified version of the formula syntax is used to describe the regression task. When an `indsizelimit` is given, individuals exceeding this limit will receive a fitness of `Inf`.

## Usage

```
makeRegressionFitnessFunction(formula, data, envir, errorMeasure = rmse,
  indsizelimit = NA, penalizeGenotypeConstantIndividuals = FALSE,
  subSamplingShare = 1)
```

## Arguments

| | |
|---|---|
| `formula` | A formula object describing the regression task. |
| `data` | An optional data frame containing the variables in the model. |
| `envir` | The R environment to evaluate individuals in. |
| `errorMeasure` | A function to use as an error measure, defaults to RMSE. |
| `indsizelimit` | Individuals exceeding this size limit will get a fitness of `Inf`. |
| `penalizeGenotypeConstantIndividuals` | Individuals that do not contain any input variables will get a fitness of `Inf`. |

| | |
|---|---|
| `subSamplingShare` | The share of fitness cases $$s$$ sampled for evaluation with each function evaluation. $$0 < s \leq 1$$ must hold, defaults to `1.0`. |

## Value

A fitness function to be used in symbolic regression.

---

`makeSeSymbolicFitnessFunction`
*Create a fitness function based on symbolic squared error (SE)*

---

## Description

Creates a fitness function that calculates the squared error of an individual with respect to a reference function `func`. When an `indsizelimit` is given, individuals exceeding this limit will receive a fitness of `Inf`.

## Usage

```
makeSeSymbolicFitnessFunction(func, lower, upper, subdivisions = 100,
  indsizelimit = NA)
```

## Arguments

| | |
|---|---|
| `func` | The reference function. |
| `lower` | The lower limit of integraion. |
| `upper` | The upper limit of integraion. |
| `subdivisions` | The maximum number of subintervals for numeric integration. |
| `indsizelimit` | Individuals exceeding this size limit will get a fitness of `Inf`. |

## Value

A fitness function based on the reference function `func`.

`makeStepsStopCondition` *Evolution stop conditions*

## Description

Evolution stop conditions are predicates (functions that return a single logical value) of the signature `function(population, stepNumber, evaluationNumber, bestFitness,timeElapsed)`. They are used to decide when to finish a GP evolution run. Stop conditions must be members of the S3 class `c("stopCondition", "function")`. They can be combined using the functions `andStopCondition`, `orStopCondition` and `notStopCondition`.

## Usage

```
makeStepsStopCondition(stepLimit)

makeEvaluationsStopCondition(evaluationLimit)

makeFitnessStopCondition(fitnessLimit)

makeTimeStopCondition(timeLimit)

andStopCondition(e1, e2)

orStopCondition(e1, e2)

notStopCondition(e1)
```

## Arguments

| | |
|---|---|
| `stepLimit` | The maximum number of evolution steps for `makeStepsStopCondition`. |
| `evaluationLimit` | The maximum number of fitness function evaluations for `makeEvaluationsStopCondition`. |
| `fitnessLimit` | The minimum fitness for `makeFitnessStopCondition`. |
| `timeLimit` | The maximum runtime in seconds for `makeTimeStopCondition`. |
| `e1` | A stop condition. |
| `e2` | A stop condition. |

## Details

`makeStepsStopCondition` creates a stop condition that is fulfilled if the number of evolution steps exceeds a given limit. `makeEvaluationsStopCondition` creates a stop condition that is fulfilled if the number of fitness function evaluations exceeds a given limit. `makeFitnessStopCondition` creates a stop condition that is fulfilled if the number best fitness seen in an evaluation run undercuts a certain limit. `makeTimeStopCondition` creates a stop condition that is fulfilled if the run time (in seconds) of an evolution run exceeds a given limit.

## `makeTinyGpSearchHeuristic` *Tiny GP Search Heuristic for RGP*

### Description

The search-heuristic, i.e. the concrete GP search algorithm, is a modular component of RGP. `makeTinyGpSearchHeuristic` creates an RGP search-heuristic that mimics the search heuristic implemented in Riccardo Poli's TinyGP system.

### Usage

```
makeTinyGpSearchHeuristic(crossoverProbability = 0.9, tournamentSize = 2)
```

### Arguments

| | |
|---|---|
| `crossoverProbability` | The crossover probability for search-heuristics that support this setting (i.e. TinyGP). Defaults to `0.9`. |
| `tournamentSize` | The size of TinyGP's selection tournaments. |

### Value

An RGP search heuristic.

## `makeTournamentSelection` *GP selection functions*

### Description

A GP selection function determines which individuals in a population should survive, i.e. are selected for variation or cloning, and which individuals of a population should be replaced. Single-objective selection functions base their selection decision on scalar fitness function, whereas multi-objective selection functions support vector-valued fitness functions. Every selection function takes a population and a (possibly vector-valued) fitness function as required arguments. It returns a list of two tables `selected` and `discarded`, with columns `index` and `fitness` each. The returned list also contains a single integer `numberOfFitnessEvaluations` that contains the number of fitness evaluations used to make the selection (Note that in the multi-objective case, evaluating all fitness functions once counts as a single evaluation). The first table contains the population indices of the individuals selected as survivors, the second table contains the population indices of the individuals that should be discarded and replaced. This definition simplifies the implementation of *steady-state* evolutionary strategies where most of the individuals in a population are unchanged in each selection step. In a GP context, steady-state strategies are often more efficient than generational strategies.

## Usage

```
makeTournamentSelection(tournamentSize = 10,
  selectionSize = ceiling(tournamentSize/2), tournamentDeterminism = 1,
  vectorizedFitness = FALSE)

makeMultiObjectiveTournamentSelection(tournamentSize = 30,
  selectionSize = ceiling(tournamentSize/2), tournamentDeterminism = 1,
  vectorizedFitness = FALSE,
  rankingStrategy = orderByParetoCrowdingDistance)

makeComplexityTournamentSelection(tournamentSize = 30,
  selectionSize = ceiling(tournamentSize/2), tournamentDeterminism = 1,
  vectorizedFitness = FALSE,
  rankingStrategy = orderByParetoCrowdingDistance,
  complexityMeasure = fastFuncVisitationLength)
```

## Arguments

| | |
|---|---|
| `complexityMeasure` | The function used to measure the complexity of an individual. |
| `tournamentSize` | The number of individuals to randomly select to form a tournament, defaults to 10 in the single-objective case, 30 in the multi-objective case. |
| `selectionSize` | The number of individuals to return as selected. |
| `tournamentDeterminism` | The propability *p* for selecting the best individual in a tournament, must be in the interval (0.0, 1.0]. The best individual is selected with propability *p*, the second best individual is selected with propability *p * (1 - p)*, the third best individual ist selected with propability *p * (1 - p)^2*, and so on. Note that setting `tournamentDeterminism` to `1.0` (the default) yields determistic behavior. |
| `vectorizedFitness` | If `TRUE`, the fitness function is expected to take a list of individuals as input and return a list of (possible vector-valued) fitnesses as output. |
| `rankingStrategy` | The strategy used to rank individuals based on multiple objectives. This function must turn a fitness vector (one point per column) into an ordering permutation (similar to the one returned by `order`). Defaults to `orderByParetoCrowdingDistance`. |

## Details

`makeTournamentSelection` returns a classic single-objective tournament selection function. `makeMultiObjectiveTournam` returns a multi-objective tournament selection function that selects individuals based on multiple objectives. `makeComplexityTournamentSelection` returns a multi-objective selection function that implements the common case of dual-objective tournament selection with high solution quality as the first objective and low solution complexity as the second objective.

## Value

A selection function.

## MapExpressionNodes *Common higher-order functions for transforming R expressions*

### Description

MapExpressionNodes transforms an expression expr by replacing every node in the tree with the result of applying a function f. The parameters functions, inners, and leafs control if f should be applied to the function symbols, inner subtrees, and leafs of expr, respectively. MapExpressionLeafs and MapExpressionSubtrees are shorthands for calls to MapExpressionNodes. expr. an expression expr. expr, given as list of nodes and and list of vertices. Each vertex is represented as a pair of indices into the list of nodes. AllExpressionNodes checks if all nodes in the tree of expr satisfy the predicate p (p returns TRUE for every node). This function short-cuts returning FALSE as soon as a node that does not satisfy p is encountered. AnyExpressionNode checks if any node in the tree of expr satisfies the predicate p. This function short-cuts returning TRUE as soon as a node that satisfies p is encountered. subtreeAt returns the subtree at index. replaceSubtreeAt replaces the subtree at index with replacement and returns the result.

### Usage

```
MapExpressionNodes(f, expr, functions = TRUE, inners = FALSE,
  leafs = TRUE)

MapExpressionLeafs(f, expr)

MapExpressionSubtrees(f, expr)

FlattenExpression(expr)

subtrees(expr, functions = FALSE, inners = TRUE, leafs = TRUE)

expressionGraph(expr)

AllExpressionNodes(p, expr)

AnyExpressionNode(p, expr)

subtreeAt(expr, index)

replaceSubtreeAt(expr, index, replacement)
```

### Arguments

| | |
|---|---|
| f | The function to apply. |
| functions | Whether to apply f to the function symbols of expr. Defaults to TRUE. |
| inners | Whether to apply f to the inner subtrees of expr. Defaults to FALSE. |
| leafs | Wheter to apply f to the leafs of expr. Defaults to TRUE. |

| | |
|---|---|
| `p` | The predicate to check. |
| `expr` | The expression to transform. |
| `index` | An in-order subtree index starting from `0` (the root). |
| `replacement` | An expression. |

### Value

The transformed expression.

---

## `mse` *Mean squared error (MSE)*

---

### Description

Mean squared error (MSE)

### Usage

```
mse(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The MSE between `x` and `y`.

---

## `multiNicheGeneticProgramming` *Cluster-based multi-niche genetic programming*

---

### Description

Perform a multi-niche genetic programming run. The required argument `fitnessFunction` must be supplied with an objective function that assigns a numerical fitness value to an R function. Fitness values are minimized, i.e. smaller values mean higher/better fitness. If a multi-objective `selectionFunction` is used, `fitnessFunction` return a numerical vector of fitness values. In a multi-niche genetic programming run, the initial population is clustered via a `clusterFunction` into `numberOfNiches` niches. In each niche, a genetic programming run is executed with `passStopCondition` as stop condition. These runs are referred to as a parallel pass. After each parallel pass, the niches are joined again using a `joinFunction` into a population. From here, the process starts again with a clustering step, until the global `stopCondition` is met. The result of the multi-niche genetic programming run is a genetic programming result object containing a GP population of R functions. `summary.geneticProgrammingResult` can be used to create summary views of a GP result object.

## Usage

```
multiNicheGeneticProgramming(fitnessFunction,
  stopCondition = makeTimeStopCondition(25),
  passStopCondition = makeTimeStopCondition(5), numberOfNiches = 2,
  clusterFunction = groupListConsecutive, joinFunction = function(niches)
  Reduce(c, niches), population = NULL, populationSize = 100,
  eliteSize = ceiling(0.1 * populationSize), elite = list(),
  functionSet = mathFunctionSet, inputVariables = inputVariableSet("x"),
  constantSet = numericConstantSet, crossoverFunction = crossover,
  mutationFunction = NULL, restartCondition = makeEmptyRestartCondition(),
  restartStrategy = makeLocalRestartStrategy(),
  searchHeuristic = makeAgeFitnessComplexityParetoGpSearchHeuristic(),
  progressMonitor = NULL, verbose = TRUE, clusterApply = sfClusterApplyLB,
  clusterExport = sfExport)
```

## Arguments

| | |
|---|---|
| `fitnessFunction` | In case of a single-objective selection function, `fitnessFunction` must be a single function that assigns a numerical fitness value to a GP individual represented as a R function. Smaller fitness values mean higher/better fitness. If a multi-objective selection function is used, `fitnessFunction` must return a numerical vector of fitness values. |
| `stopCondition` | The stop condition for the evolution main loop. See `makeStepsStopCondition` for details. |
| `passStopCondition` | The stop condition for each parallel pass. See `makeStepsStopCondition` for details. |
| `numberOfNiches` | The number of niches to cluster the population into. |
| `clusterFunction` | The function used to cluster the population into niches. The first parameter of this function is a GP population, the second paramater an integer representing the number of niches. Defaults to `groupListConsecutive`. |
| `joinFunction` | The function used to join all niches into a population again after a round of parallel passes. Defaults to a function that simply concatenates all niches. |
| `population` | The GP population to start the run with. If this parameter is missing, a new GP population of size `populationSize` is created through random growth. |
| `populationSize` | The number of individuals if a population is to be created. |
| `eliteSize` | The number of "elite" individuals to keep. Defaults to `ceiling(0.1 * populationSize)`. |
| `elite` | The elite list, must be alist of individuals sorted in ascending order by their first fitness component. |
| `functionSet` | The function set. |
| `inputVariables` | The input variable set. |
| `constantSet` | The set of constant factory functions. |

| | |
|---|---|
| `searchHeuristic` | The search-heuristic (i.e. optimization algorithm) to use in the search of solutions. See the documentation for `searchHeuristics` for available algorithms. |
| `crossoverFunction` | The crossover function. |
| `mutationFunction` | The mutation function. |
| `restartCondition` | The restart condition for the evolution main loop. See makeFitnessStagnationRestartCondition for details. |
| `restartStrategy` | The strategy for doing restarts. See makeLocalRestartStrategy for details. |
| `progressMonitor` | A function of signature `function(population, objectiveVectors, fitnessFunction, stepNumber` to be called with each evolution step. Seach heuristics may pass additional information via the `...` parameter. |
| `verbose` | Whether to print progress messages. |
| `clusterApply` | The cluster apply function that is used to distribute the parallel passes to CPUs in a compute cluster. |
| `clusterExport` | A function that is used to export R variables to the nodes of a CPU cluster, defaults to `sfExport`. |

## Value

A genetic programming result object that contains a GP population in the field `population`, as well as metadata describing the run parameters.

## See Also

`geneticProgramming`, `summary.geneticProgrammingResult`, `symbolicRegression`

---

## `multiNicheSymbolicRegression`

*Symbolic regression via multi-niche standard genetic programming*

---

## Description

Perform symbolic regression via untyped multi-niche genetic programming. The regression task is specified as a `formula`. Only simple formulas without interactions are supported. The result of the symbolic regression run is a symbolic regression model containing an untyped GP population of model functions.

## Usage

```
multiNicheSymbolicRegression(formula, data,
  stopCondition = makeTimeStopCondition(25),
  passStopCondition = makeTimeStopCondition(5), numberOfNiches = 2,
  clusterFunction = groupListConsecutive, joinFunction = function(niches)
  Reduce(c, niches), population = NULL, populationSize = 100,
  eliteSize = ceiling(0.1 * populationSize), elite = list(),
  individualSizeLimit = 64, penalizeGenotypeConstantIndividuals = FALSE,
  functionSet = mathFunctionSet, constantSet = numericConstantSet,
  selectionFunction = makeTournamentSelection(),
  crossoverFunction = crossover, mutationFunction = NULL,
  restartCondition = makeEmptyRestartCondition(),
  restartStrategy = makeLocalRestartStrategy(), progressMonitor = NULL,
  verbose = TRUE, clusterApply = sfClusterApplyLB,
  clusterExport = sfExport)
```

## Arguments

| | |
|---|---|
| `formula` | A `formula` describing the regression task. Only simple formulas of the form `response ~ variable1 + ... + variableN` are supported at this point in time. |
| `data` | A `data.frame` containing training data for the symbolic regression run. The variables in `formula` must match column names in this data frame. |
| `stopCondition` | The stop condition for the evolution main loop. See `makeStepsStopCondition` for details. |
| `passStopCondition` | The stop condition for each parallel pass. See `makeStepsStopCondition` for details. |
| `numberOfNiches` | The number of niches to cluster the population into. |
| `clusterFunction` | The function used to cluster the population into niches. The first parameter of this function is a GP population, the second paramater an integer representing the number of niches. Defaults to `groupListConsecutive`. |
| `joinFunction` | The function used to join all niches into a population again after a round of parallel passes. Defaults to a function that simply concatenates all niches. |
| `population` | The GP population to start the run with. If this parameter is missing, a new GP population of size `populationSize` is created through random growth. |
| `populationSize` | The number of individuals if a population is to be created. |
| `eliteSize` | The number of "elite" individuals to keep. Defaults to `ceiling(0.1 * populationSize)`. |
| `elite` | The elite list, must be alist of individuals sorted in ascending order by their first fitness component. |
| `individualSizeLimit` | Individuals with a number of tree nodes that exceeds this size limit will get a fitness of `Inf`. |
| `penalizeGenotypeConstantIndividuals` | Individuals that do not contain any input variables will get a fitness of `Inf`. |

| | |
|---|---|
| `functionSet` | The function set. |
| `constantSet` | The set of constant factory functions. |
| `selectionFunction` | The selection function to use. Defaults to tournament selection. See makeTournamentSelection for details. |
| `crossoverFunction` | The crossover function. |
| `mutationFunction` | The mutation function. |
| `restartCondition` | The restart condition for the evolution main loop. See makeFitnessStagnationRestartCondition for details. |
| `restartStrategy` | The strategy for doing restarts. See makeLocalRestartStrategy for details. |
| `progressMonitor` | A function of signature `function(population, objectiveVectors, fitnessFunction, stepNumber` to be called with each evolution step. Seach heuristics may pass additional information via the `...` parameter. |
| `verbose` | Whether to print progress messages. |
| `clusterApply` | The cluster apply function that is used to distribute the parallel passes to CPUs in a compute cluster. |
| `clusterExport` | A function that is used to export R variables to the nodes of a CPU cluster, defaults to snowfall's `sfExport`. |

## Value

An symbolic regression model that contains an untyped GP population.

## See Also

`predict.symbolicRegressionModel`, `geneticProgramming`

---

`mutateFunc` *Random mutation of functions and expressions*

---

## Description

RGP implements two sets of mutation operators. The first set is inspired by classical GP systems. Mutation strength is controlled by giving mutation probabilities: `mutateFunc` mutates a function $f$ by recursively replacing inner function labels in $f$ with probability `mutatefuncprob`. `mutateSubtree` mutates a function by recursively replacing inner nodes with newly grown subtrees of maximum depth `maxsubtreedepth`. `mutateNumericConst` mutates a function by perturbing each numeric (double) constant $c$ with probability `mutateconstprob` by setting $c := c + rnorm(1, mean = mu, sd = sigma)$. Note that constants of other typed than `double` (e.g `integers`) are not affected.

## Usage

```
mutateFunc(func, funcset, mutatefuncprob = 0.1,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

mutateSubtree(func, funcset, inset, conset, mutatesubtreeprob = 0.1,
  maxsubtreedepth = 5, breedingFitness = function(individual) TRUE,
  breedingTries = 50)

mutateNumericConst(func, mutateconstprob = 0.1,
  breedingFitness = function(individual) TRUE, breedingTries = 50, mu = 0,
  sigma = 1)

mutateFuncTyped(func, funcset, mutatefuncprob = 0.1,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

mutateSubtreeTyped(func, funcset, inset, conset, mutatesubtreeprob = 0.1,
  maxsubtreedepth = 5, breedingFitness = function(individual) TRUE,
  breedingTries = 50)

mutateNumericConstTyped(func, mutateconstprob = 0.1,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

mutateChangeLabel(func, funcset, inset, conset, strength = 1,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

mutateInsertSubtree(func, funcset, inset, conset, strength = 1,
  subtreeDepth = 2, breedingFitness = function(individual) TRUE,
  breedingTries = 50)

mutateDeleteSubtree(func, funcset, inset, conset, strength = 1,
  subtreeDepth = 2, constprob = 0.2,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

mutateChangeDeleteInsert(func, funcset, inset, conset, strength = 1,
  subtreeDepth = 2, constprob = 0.2, iterations = 1,
  changeProbability = 1/3, deleteProbability = 1/3,
  insertProbability = 1/3, breedingFitness = function(individual) TRUE,
  breedingTries = 50)

mutateDeleteInsert(func, funcset, inset, conset, strength = 1,
  subtreeDepth = 2, constprob = 0.2, iterations = 1,
  deleteProbability = 0.5, insertProbability = 0.5,
  breedingFitness = function(individual) TRUE, breedingTries = 50)

mutateFuncFast(funcbody, funcset, mutatefuncprob = 0.1)

mutateSubtreeFast(funcbody, funcset, inset, constmin, constmax, insertprob,
  deleteprob, subtreeprob, constprob, maxsubtreedepth)
```

```
mutateNumericConstFast(funcbody, mutateconstprob = 0.1, mu = 0, sigma = 1)
```

## Arguments

| | |
|---|---|
| `func` | The function to mutate randomly. |
| `funcbody` | The function body to mutate randomly, obtain it via `body(func)`. |
| `funcset` | The function set. |
| `inset` | The set of input variables. |
| `conset` | The set of constant factories. |
| `mutatefuncprob` | The probability of trying to replace an inner function at each node. |
| `mutatesubtreeprob` | The probability of replacing a subtree with a newly grown subtree at each node. |
| `maxsubtreedepth` | The maximum depth of newly grown subtrees. |
| `mutateconstprob` | The probability of mutating a constant by adding `rnorm(1)` to it. |
| `strength` | The number of individual point mutations (changes, insertions, deletions) to perform. |
| `subtreeDepth` | The depth of the subtrees to insert or delete. |
| `constprob` | The probability of creating a constant versus an input variable. |
| `insertprob` | The probability to insert a subtree. |
| `deleteprob` | The probability to insert a subtree. |
| `constmin` | The lower limit for numeric constants. |
| `constmax` | The upper limit for numeric onstants. |
| `mu` | The normal distribution mean for random numeric constant mutation. |
| `sigma` | The normal distribution standard deviation for random numeric constant mutation. |
| `subtreeprob` | The probability of creating a subtree instead of a leaf in the random subtree generator function. |
| `iterations` | The number of times to apply a mutation operator to a GP individual. This can be used as a generic way of controling the strength of the genotypic effect of mutation. |
| `changeProbability` | The probability for selecting the `mutateChangeLabel` operator. |
| `deleteProbability` | The probability for selecting the `mutateDeleteSubtree` operator. |
| `insertProbability` | The probability for selecting the `mutateInsertSubtree` operator. |
| `breedingFitness` | A breeding function. See the documentation for `geneticProgramming` for details. |
| `breedingTries` | The number of breeding steps. |

## Details

`mutateFuncTyped`, `mutateSubtreeTyped`, and `mutateNumericConstTyped` are variants of the above functions that only create well-typed result expressions.

`mutateFuncFast`, `mutateSubtreeFast`, `mutateNumericConstFast` are variants of the above untyped mutation function implemented in C. They offer a considerably faster execution speed for the price of limited flexibility. These variants take function bodies as arguments (obtain these via R's body function) and return function bodies as results. To turn a function body into a function, use RGP's `makeClosure` tool function.

The second set of mutation operators features a more orthogonal design, with each individual operator having a only a small effect on the genotype. Mutation strength is controlled by the integral `strength` parameter. `mutateChangeLabel` Selects a node (inner node or leaf) by uniform random sampling and replaces the label of this node by a new label of matching type. `mutateInsertSubtree` Selects a leaf by uniform random sampling and replaces it with a matching subtree of the exact depth of `subtreeDepth`. `mutateDeleteSubtree` Selects a subree of the exact depth of `subtreeDepth` by uniform random sampling and replaces it with a matching leaf. `mutateChangeDeleteInsert` Either applies `mutateChangeLabel`, `mutateInsertSubtree`, or `mutateDeleteSubtree`. The probability weights for selecting an operator can be supplied via the ...Probability arguments (probability weights are normalized to a sum of 1). `mutateDeleteInsert` Either applies `mutateDeleteSubtree` or `mutateInsertSubtree`. The probability weights for selecting an operator can be supplied via the ...Probability arguments (probability weights are normalized to a sum of 1). The above functions automatically create well-typed result expressions when used in a strongly typed GP run.

All RGP mutation operators have the S3 class `c("mutationOperator", "function")`.

## Value

The randomly mutated function.

---

`new.alist` *Create a new function argument list from a list or vector of strings*

---

## Description

Creates a formal argument list from a list or vector of strings, ready to be assigned via `formals`.

## Usage

```
new.alist(fargs)
```

## Arguments

| | |
|---|---|
| `fargs` | The formal arguments, given as a list or vector of strings. |

## Value

A formal argument list, ready to be passed via `formals`.

## `new.function` *Create a new function stub*

### Description

Creates and returns a new function stub without capturing any environment variables.

### Usage

```
new.function(envir = globalenv())
```

### Arguments

| | |
|---|---|
| `envir` | The new function closure's environment, defaults to `globalenv()`. |

### Value

A new function that does not take any arguments and always returns `NULL`.

### Note

Always use this function to dynamically generate new functions that are not clojures to prevent hard to find memory leaks.

## `nmse` *Normalized mean squared error (NMSE)*

### Description

Calculates the MSE between vectors after normalizing them into the interval [0, 1].

### Usage

```
nmse(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The NMSE between `x` and `y`.

## `nondeterministicRanking` *Create a nondeterministic ranking*

### Description

Create a permutation of the sequence `s = 1:l` representing a ranking. If `p` = 1, the ranking will be completely deterministic, i.e. equal to `1:l`. If `p` = 0, the ranking will be completely random. If 0 < `p` < 1, the places in the ranking will be determined by iterative weighted sampling without replacement from the sequence `s` := `1:l`. At each step of this iterated weighted sampling, the first remaining element of `s` will be selected with probability `p`, the second element with probability `p * (1 - p)`, the third element with probability `p * (1 - p) ^ 2`, and so forth.

### Usage

```
nondeterministicRanking(l, p = 1)
```

### Arguments

| | |
|---|---|
| `l` | The numer of elements in the ranking. |
| `p` | The "degree of determinism" of the ranking to create. |

### Value

A ranking permutation of the values `1:l`.

## `normalize` *Normalize a vector into the interval [0, 1]*

### Description

Normalize a vector into the interval [0, 1]

### Usage

```
normalize(x)
```

### Arguments

| | |
|---|---|
| `x` | The vector to normalize, so that each element lies in the interval [0, 1]. |

### Value

The normalized vector.

## `normalizedDesign` *Create a normalized design matrix*

### Description

Produces a normalized design and calculates the minimal distance if required. Returns a design is a matrix with `dim` columns and `size` rows.

### Usage

```
normalizedDesign(dimension, size, calcMinDistance = FALSE)
```

### Arguments

| | |
|---|---|
| `dimension` | Dimension of the problem (will be no. of columns of the result matrix). |
| `size` | Number of points with that dimension needed. (will be no. of rows of the result matrix). |
| `calcMinDistance` | Indicates whether a minimal distance should be calculated. |

### Value

List `L` consists of a matrix and nd (if required) a minimal distance.

## `orderByParetoCrowdingDistance` *Rearrange points via Pareto-based rankings*

### Description

Returns a permutation that rearranges points, given as columns in a value matrix, via Pareto-based ranking. Points are ranked by their Pareto front number. In `orderByParetoCrowdingDistance`, ties are then broken by crowding distance, in `orderByParetoHypervolumeContribution`, ties are broken by hypervolume contribution.

### Usage

```
orderByParetoCrowdingDistance(values)

orderByParetoHypervolumeContribution(values)
```

### Arguments

| | |
|---|---|
| `values` | The value matrix to return the ordering permutation for. Each column represents a point, each row a dimension. |

### Value

A permutation to rearrange `values` based on a Pareto based ranking.

---

## `orderByParetoMeasure` *Rearrange points via an arbitrary Pareto-based ranking*

---

### Description

Returns a permutation that rearranges points, given as columns in a value matrix, via Pareto-based ranking. Points are ranked by their Pareto front number, ties are broken by the values of `measure`.

### Usage

```
orderByParetoMeasure(values, measure = crowding_distance)
```

### Arguments

| | |
|---|---|
| `values` | The value matrix to return the ordering permutation for. Each column represents a point, each row a dimension. |
| `measure` | The measure used for ranking points that lie on the same Pareto front, defaults to `crowding_distance`. |

### Value

A permutation to rearrange `values` based on a Pareto based ranking.

---

## `paretoFrontKneeIndex` *Find the knee of a two dimensional pareto front*

---

### Description

Given a matrix `m` of two rows and n columns, representing solutions of a two-dimensional optimization problem, returns the column index of the point with minimum euclidean distance to the utopia point. The utopia point is the point consisting of the row minima of `m`. `NA` or `NaN` values of `m` are ommited.

### Usage

```
paretoFrontKneeIndex(m, normalize = TRUE)
```

### Arguments

| | |
|---|---|
| `m` | A matrix of two rows and n columns, representing the solutions of a two-dimensional optimization problem. |
| `normalize` | Whether to normalize both objectives to the interval of [0, 1], defaults to `TRUE`. |

## Value

The knee point index, i.e. the column index in m of the point of minimum euclidean distance to the utopia point.

## Examples

```
m1 <- matrix(runif(200), ncol = 100)
plot(t(m1))
points(t(m1[,emoa::nds_rank(m1) == 1]), col = "red", pch = 16)
pKnee <- m1[, paretoFrontKneeIndex(m1)]
points(t(pKnee), col = "green4", pch = 16)
```

---

`plotFunction3d` *Plot a 2D function as a 3D surface*

---

## Description

Creates and shows and perspective plot of a 2D function of either the form $z = f(x, y)$ or $z = f(xv)$, where $xv$ is a numeric of length 2.

## Usage

```
plotFunction3d(func = function(x) sum(x^2), lo = c(0, 0), up = c(1, 1),
  samples = 10, palette = gray.colors(256), ...)
```

## Arguments

| | |
|---|---|
| `func` | A 2D function to plot. |
| `lo` | A vector of lower limits of the plot (one entry for each dimension). |
| `up` | A vector of upper limits of the plot (one entry for each dimension). |
| `samples` | The number of samples in each dimension. |
| `palette` | The color palette, use `NULL` to disable. |
| `...` | Graphic parameters for `persp`. |

`plotFunctions` *Show an overlayed plot of multiple functions*

## Description

Creates and shows and overlayed plot of one or more functions of one variable $y = f(x)$.

## Usage

```
plotFunctions(funcs, from = 0, to = 1, steps = 1024, type = "l",
  lty = 1:5, lwd = 1, lend = par("lend"), pch = NULL, col = 1:6,
  cex = NULL, bg = NA, xlab = "x", ylab = "y",
  legendpos = "bottomright", bty = "n", ...)
```

## Arguments

| | |
|---|---|
| `funcs` | A list of functions of one variable to plot. |
| `from` | The left bound of the plot, i.e. the minimum $x$ value to plot. |
| `to` | The right bound of the plot, i.e. the maximum $x$ value to plot. |
| `steps` | The number of steps, or samples, to plot. |
| `type` | The plot type (e.g. l = line) as passed on to `matplot`. |
| `lty` | The line types as passed on to `matplot`. |
| `lwd` | The line widths as passed on to `matplot`. |
| `lend` | The line end cap types as passed on to `matplot`. |
| `pch` | The plot chars as passed on to `matplot`. |
| `col` | The plot colors as passed on to `matplot`. |
| `cex` | The character expansion sizes as passed on to `matplot`. |
| `bg` | The background (fill) colors as passed on to `matplot`. |
| `xlab` | The x axis label as passed on to `matplot`. |
| `ylab` | The y axis label as passed on to `matplot`. |
| `legendpos` | The position of the legend, passed as the `x` parameter to `legend`. |
| `bty` | The box type parameter of the legend, passed as the `bty` parameter to `legend`. |
| `...` | Graphic parameters for `par` and further arguments to `plot`. For example, use the `main` parameter to set a title. |

## Examples

```
plotFunctions(list(function(x) sin(x),
                   function(x) cos(x),
                   function(x) 0.5*sin(2*x)+1),
              -pi, pi, 256)
```

## plotParetoFront *Plot a GP Pareto Front*

### Description

Plots fitness/complexity/age Pareto fronts for multi-objective GP. The z-coordinate represents individual age and is shown in form of a color scale, where younger individuals are bright green, individuals with age maxZ are black. Individuals not on the first Pareto front are shown as small gray circles, regardless of age.

### Usage

```
plotParetoFront(x, y, z, indicesToMark = integer(), maxZ = 50,
  main = sprintf("Population Pareto Front Plot (% Individuals)", length(x)),
  ...)
```

### Arguments

| | |
|---|---|
| x | A vector of type numeric representing individual fitness. |
| y | A vector of type numeric representing individual complexity. |
| z | A vector of type integer representing individual age. |
| indicesToMark | A index vector of points to mark with red crosses. |
| maxZ | The individual age at the large end of the age color scale. |
| main | The plot's title. |
| ... | Graphic parameters for par and further arguments to plot. For example, use the main parameter to set a title. |

### See Also

funcToIgraph

## plotPopulationFitnessComplexity *Fitness/Complexity plot for populations*

### Description

Plots the fitness against the complexity of each individual in a population.

### Usage

```
plotPopulationFitnessComplexity(pop, fitnessFunction,
  complexityFunction = fastFuncVisitationLength, showIndices = TRUE,
  showParetoFront = TRUE, hideOutliers = 0, ...)
```

## Arguments

| | |
|---|---|
| `pop` | A population to plot. |
| `fitnessFunction` | The function to calculate an individual's fitness with. |
| `complexityFunction` | The function to calculate an individual's complexity with. |
| `showIndices` | Whether to show the population index of each individual. |
| `showParetoFront` | Whether to highlight the pareto front in the plot. |
| `hideOutliers` | If `N = hideOutliers > 0`, hide outliers from the plot using a "N * IQR" criterion. |
| `...` | Additional parameters for the underlying call to `plot`. |

---

`popfitness` *Calculate the fitness value of each individual in a population*

---

## Description

Calculate the fitness value of each individual in a population

## Usage

```
popfitness(pop, fitnessfunc)
```

## Arguments

| | |
|---|---|
| `pop` | A population of functions. |
| `fitnessfunc` | The fitness function. |

## Value

A list of fitness function values in the same order as `pop`.

## predict.symbolicRegressionModel

*Predict method for symbolic regression models*

### Description

Predict values via a model function from a population of model functions generated by symbolic regression.

### Usage

```
## S3 method for class 'symbolicRegressionModel'
predict(object, newdata, model = "BEST",
  detailed = FALSE, ...)
```

### Arguments

| | |
|---|---|
| `object` | A model created by `symbolicRegression`. |
| `newdata` | A `data.frame` containing input data for the symbolic regression model. The variables in `object$formula` must match column names in this data frame. |
| `model` | The numeric index of the model function in `object$population` to use for prediction or `"BEST"` to use the model function with the best training fitness. |
| `detailed` | Whether to add metadata to the prediction object returned. |
| `...` | Ignored in this `predict` method. |

### Value

A vector of predicted values or, if `detailed` is `TRUE`, a list of the following elements: `model` the model used in this prediction `response` a matrix of predicted versus respone values `RMSE` the RMSE between the real and predicted response

## print.sType

*Prints a sType and returns it invisible.*

### Description

Prints a sType and returns it invisible.

### Usage

```
## S3 method for class 'sType'
print(x, ...)
```

### Arguments

| | |
|---|---|
| `x` | The sType to print. |
| `...` | Optional parameters to print are ignored in this method. |

## `randchild` *Select random childs or subtrees of an expression*

### Description

`randchild` returns a uniformly random direct child of an expression. `randsubtree` returns a uniformly random subtree of an expression. Note that this subtree must not be a direct child.

### Usage

```
randchild(expr)

randsubtree(expr, subtreeprob = 0.1)
```

### Arguments

| | |
|---|---|
| `expr` | The expression to select random childs or subtrees from. |
| `subtreeprob` | The probability for `randsubtree` to select a certain subtree instead of searching further via an recursive call. |

## `randelt` *Choose a random element from a list or vector*

### Description

Returns a unformly random chosen element of the vector or list `x`.

### Usage

```
randelt(x, prob = NULL)
```

### Arguments

| | |
|---|---|
| `x` | The vector or list to chose an element from. |
| `prob` | A vector of probability weights for obtaining the elements of the vector or list being sampled. |

### Value

A uniformly random element of `x`.

## randexprGrow *Creates an R expression by random growth*

### Description

Creates a random R expression by randomly growing its tree. In each step of growth, with probability `subtreeprob`, an operator is chosen from the function set `funcset`. The operands are then generated by recursive calls. If no subtree is generated, a constant will be generated with probability `constprob`. If no constant is generated, an input variable will be chosen randomly. The depth of the resulting expression trees can be bounded by the `maxdepth` parameter. `randexprFull` creates a random full expression tree of depth `maxdepth`. The algorithm is the same as `randexprGrow`, with the exception that the probability of generating a subtree is fixed to 1 until the desired tree depth `maxdepth` is reached.

### Usage

```
randexprGrow(funcset, inset, conset, maxdepth = 8, constprob = 0.2,
  subtreeprob = 0.5, curdepth = 1)

randexprFull(funcset, inset, conset, maxdepth = 8, constprob = 0.2)
```

### Arguments

| | |
|---|---|
| `funcset` | The function set. |
| `inset` | The set of input variables. |
| `conset` | The set of constant factories. |
| `maxdepth` | The maximum expression tree depth. |
| `constprob` | The probability of generating a constant in a step of growth, if no subtree is generated. If neither a subtree nor a constant is generated, a randomly chosen input variable will be generated. Defaults to `0.2`. |
| `subtreeprob` | The probability of generating a subtree in a step of growth. |
| `curdepth` | (internal) The depth of the random expression currently generated, used internally in recursive calls. |

### Value

A new R expression generated by random growth.

`randexprTypedGrow` *Creates an R expression by random growth respecting type constraints*

## Description

Creates a random R expression by randomly growing its tree. In each step of growth, with probability `subtreeprob`, an operator is chosen from the function set `funcset`. The operands are then generated by recursive calls. If no function of matching range type exists, a terminal (constant or input variable) will be generated instead. If no subtree is generated, a constant will be generated with probability `constprob`. If no constant is generated, an input variable will be chosen randomly. The depth of the resulting expression trees can be bounded by the `maxdepth` parameter. In contrast to `randexprGrow`, this function respects sTypes of functions, input variables, and constant factories. Only well-typed expressions are created. `randexprTypedFull` creates a random full expression tree of depth `maxdepth`, respecting type constraints.

## Usage

```
randexprTypedGrow(type, funcset, inset, conset, maxdepth = 8,
  constprob = 0.2, subtreeprob = 0.5, curdepth = 1)

randexprTypedFull(type, funcset, inset, conset, maxdepth = 8,
  constprob = 0.2)
```

## Arguments

| | |
|---|---|
| `type` | The (range) type the created expression should have. |
| `funcset` | The function set. |
| `inset` | The set of input variables. |
| `conset` | The set of constant factories. |
| `maxdepth` | The maximum expression tree depth. |
| `constprob` | The probability of generating a constant in a step of growth, if no subtree is generated. If neither a subtree nor a constant is generated, a randomly chosen input variable will be generated. Defaults to `0.2`. |
| `subtreeprob` | The probability of generating a subtree in a step of growth. |
| `curdepth` | (internal) The depth of the random expression currently generated, used internally in recursive calls. |

## Value

A new R expression generated by random growth.

## randfunc *Creates an R function with a random expression as its body*

### Description

Creates an R function with a random expression as its body

### Usage

```
randfunc(funcset, inset, conset, maxdepth = 8, constprob = 0.2,
  exprfactory = randexprGrow, breedingFitness = function(individual) TRUE,
  breedingTries = 50)

randfuncRampedHalfAndHalf(funcset, inset, conset, maxdepth = 8,
  constprob = 0.2, breedingFitness = function(individual) TRUE,
  breedingTries = 50)
```

### Arguments

| | |
|---|---|
| `funcset` | The function set. |
| `inset` | The set of input variables. |
| `conset` | The set of constant factories. |
| `maxdepth` | The maximum expression tree depth. |
| `exprfactory` | The function to use for randomly creating the function's body. |
| `constprob` | The probability of generating a constant in a step of growth, if no subtree is generated. If neither a subtree nor a constant is generated, a randomly chosen input variable will be generated. Defaults to `0.2`. |
| `breedingFitness` | A breeding function. See the documentation for `geneticProgramming` for details. |
| `breedingTries` | The number of breeding steps. |

### Value

A randomly generated R function.

## randfuncTyped — *Creates a well-typed R function with a random expression as its body*

### Description

Creates a well-typed R function with a random expression as its body

### Usage

```
randfuncTyped(type, funcset, inset, conset, maxdepth = 8, constprob = 0.2,
  exprfactory = randexprTypedGrow, breedingFitness = function(individual)
  TRUE, breedingTries = 50)

randfuncTypedRampedHalfAndHalf(type, funcset, inset, conset, maxdepth = 8,
  constprob = 0.2, breedingFitness = function(individual) TRUE,
  breedingTries = 50)
```

### Arguments

| | |
|---|---|
| `type` | The range type of the random function to create. |
| `funcset` | The function set. |
| `inset` | The set of input variables. |
| `conset` | The set of constant factories. |
| `maxdepth` | The maximum expression tree depth. |
| `constprob` | The probability of generating a constant in a step of growth, if no subtree is generated. If neither a subtree nor a constant is generated, a randomly chosen input variable will be generated. Defaults to `0.2`. |
| `exprfactory` | The function to use for randomly creating the function's body. |
| `breedingFitness` | A breeding function. See the documentation for `geneticProgramming` for details. |
| `breedingTries` | The number of breeding steps. |

### Value

A randomly generated well-typed R function.

## `randterminalTyped` *Create a random terminal node*

### Description

Create a random terminal node

### Usage

```
randterminalTyped(typeString, inset, conset, constprob)
```

### Arguments

| | |
|---|---|
| `typeString` | The string label of the type of the random terminal node to create. |
| `inset` | The set of input variables. |
| `conset` | The set of constant factories. |
| `constprob` | The probability of creating a constant versus an input variable. |

### Value

A random terminal node, i.e. an input variable or a constant.

## `rangeTypeOfType` *Return the range type if t is a function type, otherwise just return t*

### Description

Return the range type if t is a function type, otherwise just return t

### Usage

```
rangeTypeOfType(t)
```

### Arguments

| | |
|---|---|
| `t` | The type to extract the range type from. |

### Value

The range type.

## `rgpBenchmark` *Utility functions for testing and benchmarking the RGP system*

### Description

`rgpBenchmark` measures the number of fitness evaluations per second performed by `geneticProgramming`. A number of `samples` experiments are performed.

### Usage

```
rgpBenchmark(fitnessFunction = function(ind) 0, samples = 1, time = 10,
  ...)

evaluationsPerSecondBenchmark(f, samples = 1, time = 10, ...)
```

### Arguments

| | |
|---|---|
| `f` | The function under test. |
| `fitnessFunction` | The fitness function to pass to the call to `geneticProgramming`. |
| `samples` | The number of indpendent measurements to perform, defaults to 1. |
| `time` | The time in seconds a sample lasts, defaults to 10 seconds. |
| `...` | Options as passed to the function under test. |

### Details

`evaluationsPerSecondBenchmark` measures the number of times a function can be called per second in a tight loop.

### Value

The number of fitness evaluations per second performed by RGP.

## `rmse` *Root mean squared error (RMSE)*

### Description

Root mean squared error (RMSE)

### Usage

```
rmse(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The RMSE between `x` and `y`.

---

## `rsquared` *Coefficient of determination (R^2)*

---

### Description

Coefficient of determination (R^2)

### Usage

```
rsquared(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The coefficient of determination (R^2) between `x` and `y`.

---

## `r_mae` *R version of Mean absolute error (MAE)*

---

### Description

R version of Mean absolute error (MAE)

### Usage

```
r_mae(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The MAE between `x` and `y`.

## `r_sse` *R version of Sum squared error (SSE)*

### Description

R version of Sum squared error (SSE)

### Usage

```
r_sse(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The SSE between `x` and `y`.

## `r_ssse` *R version of Scaled sum squared error (sSSE)*

### Description

R version of Scaled sum squared error (sSSE)

### Usage

```
r_ssse(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The sSSE between `x` and `y`.

## safeDivide — *Some simple arithmetic and logic functions for use in GP expressions*

### Description

`safeDivide` a division operator that returns 0 if the divisor is 0. `safeLn` a natural logarithm operator that return 0 if its argument is less then 0. `ln` is the natural logarithm. `positive` returns true if its argument is greater then 0. `ifPositive` returns its second argument if its first argument is positive, otherwise its third argument. `ifThenElse` returns its second argument if its first argument is `TRUE`, otherwise its third argument.

### Usage

```
safeDivide(a, b)

safeSqroot(a)

safeLn(a)

ln(a)

positive(x)

ifPositive(x, thenbranch, elsebranch)

ifThenElse(x, thenbranch, elsebranch)
```

### Arguments

| | |
|---|---|
| `a` | A numeric value. |
| `b` | A numeric value. |
| `x` | A numeric value. |
| `thenbranch` | The element to return when `x` is `TRUE`. |
| `elsebranch` | The element to return when `x` is `FALSE`. |

## seSymbolic — *Symbolic squared error (SE)*

### Description

Given to functions `f` and `g`, returns the area the squared differences between `f` and `g` in the integration limits `lower` and `upper`.

### Usage

```
seSymbolic(f, g, lower, upper, subdivisions = 100)
```

### Arguments

| | |
|---|---|
| `f` | An R function. |
| `g` | An R function with the same formal arguments as `f`. |
| `lower` | The lower limit of integraion. |
| `upper` | The upper limit of integraion. |
| `subdivisions` | The maximum number of subintervals for numeric integration. |

### Value

The area of the squared differences between `f` and `g`, or `Inf` if integration is not possible in the limits given.

---

## `seSymbolicFunction` *Symbolic squared error function (SE)*

---

### Description

Given two functions `f` and `g`, returns a function whose body is the symbolic representation of the squared error between `f` and `g`, i.e. `function(x) (f(x) - g(x))^2`.

### Usage

```
seSymbolicFunction(f, g)
```

### Arguments

| | |
|---|---|
| `f` | An R function. |
| `g` | An R function with the same formal arguments as `f`. |

### Value

A function representing the squared error between `f` and `g`.

## smse *Scaled mean squared error (SMSE)*

### Description

Calculates the MSE between vectors after scaling them. Beware that this error measure is invariant to scaling with negative constants, i.e. the multiplicative inverse of the true functions also receives an error of 0. See http://www2.cs.uidaho.edu/~cs472_572/f11/scaledsymbolicRegression.pdf for details.

### Usage

```
smse(x, y)
```

### Arguments

| | |
|---|---|
| x | A numeric vector or list. |
| y | A numeric vector or list. |

### Value

The NMSE between x and y.

## sortBy *Sort a vector or list by the result of applying a function*

### Description

Sorts a vector or a list by the numerical result of applying the function byFunc.

### Usage

```
sortBy(xs, byFunc)
```

### Arguments

| | |
|---|---|
| xs | A vector or list. |
| byFunc | A function from elements of xs to numeric. |

### Value

The result of sorting xs by byfunc.

## `sortByRange` *Tabulate a list of functions or input variables by the range part of their sTypes*

### Description

Tabulate a list of functions or input variables by the range part of their sTypes

### Usage

```
sortByRange(x)
```

### Arguments

| | |
|---|---|
| `x` | A list of functions or input variables to sort by range sType. |

### Value

A table of the objects keyed by their range sTypes.

## `sortByRanking` *Sort a vector or list via a given ranking*

### Description

Reorders a vector or list according to a given ranking `ranking`.

### Usage

```
sortByRanking(xs, ranking = rank(xs))
```

### Arguments

| | |
|---|---|
| `xs` | The vector or list to reorder. |
| `ranking` | The ranking to sort `xs` by, defaults to `rank(xs)`. |

### Value

The result of reordering `xs` by `ranking`.

## sortByType — *Tabulate a list of functions or input variables by their sTypes*

### Description

Tabulate a list of functions or input variables by their sTypes

### Usage

```
sortByType(x)
```

### Arguments

| | |
|---|---|
| `x` | A list of functions or input variables to sort by sType. |

### Value

A table of the objects keyed by their sTypes.

## splitList — *Splitting and grouping of lists*

### Description

Functions for splitting and grouping lists into sublists. `splitList` splits a list `l` into `max(groupAssignment)` groups. The integer indices of `groupAssignment` determine in which group each element of `l` goes. `groupListConsecutive` splits `l` into `numberOfGroups` consecutive sublists (or groups). `groupListDistributed` distributes `l` into `numberOfGroups` sublists (or groups). `flatten` flattens a list `l` of lists into a flat list by concatenation. If `recursive` is `TRUE` (defaults to `FALSE`), flatten will be recursively called on each argument first. `intersperse` joins two lists `xs` and `ys` into a list of pairs containig every possible pair, i.e. `intersperse(xs, ys)` equals the product list of `xs` and `ys`. The `pairConstructor` parameter can be used to change the type of pairs returned.

### Usage

```
splitList(l, groupAssignment)

groupListConsecutive(l, numberOfGroups)

groupListDistributed(l, numberOfGroups)

flatten(l, recursive = FALSE)

intersperse(xs, ys, pairConstructor = list)
```

### Arguments

| | |
|---|---|
| `l` | A list. |
| `xs` | A list. |
| `ys` | A list. |
| `pairConstructor` | The function to use for constructing pairs, defaults to `list`. |
| `groupAssignment` | A vector of group assignment indices. |
| `numberOfGroups` | The number of groups to create, must be <= length(l) |
| `recursive` | Whether to operate recursively on sublists or vectors. |

### Value

A list of lists, where each member represents a group.

---

## `sse` *Sum squared error (SSE)*

---

### Description

Sum squared error (SSE)

### Usage

```
sse(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The SSE between `x` and `y`.

## `ssse` *Scaled sum squared error (sSSE)*

### Description

Scaled sum squared error (sSSE)

### Usage

```
ssse(x, y)
```

### Arguments

| | |
|---|---|
| `x` | A numeric vector or list. |
| `y` | A numeric vector or list. |

### Value

The sSSE between `x` and `y`.

## `st` *Type constructors for types in the Rsymbolic type system*

### Description

These functions create types for the Rsymbolic type system, called *sTypes* from here on. These functions are used mostly in literal expressions denoting sTypes. `st` creates a *base sType* from a string. A base sType is a type without any further structure. Example include `st("numeric")`, `st("character")` or `st("logical")`. `%->%` creates a *function sType*, i.e. the type of function, from a vector of argument sTypes and a result sType. A function sType has `domain` and `range` containing its argument and result types. Every sType has a `string` field containing a unambiguous string representation that can serve as a hash table key. STypes can be checked for equality via `identical`. `sObject` is the root of the sType hierarchy, i.e. the most general type.

### Usage

```
st(baseTypeName)

domainTypes %->% rangeType

sObject
```

## Arguments

| | |
|---|---|
| `baseTypeName` | The name of the base sType to create. |
| `domainTypes` | The `domain` sType of a function sType. |
| `rangeType` | The `range` sType of a function sType. |

## Format

```
List of 2
 $ base  : chr "sObject"
 $ string: chr "sObject"
 - attr(*, "class")= chr [1:3] "sBaseType" "sType" "character"
```

## Value

The created sType.

## See Also

sTypeInference

## Examples

```
st("numeric")
list(st("numeric"), st("numeric")) \%->\% st("logical")
is.sType(st("logical"))
```

---

`sType` *Inference of sTypes*

---

## Description

RGP internally infers the sTypes of compound expressions like function applications and function definitions from the sTypes of atomic expressions. The sTypes of building blocks are defined by the user via the `%::%` operator and are stored in the package-internal global variable `rgpSTypeEnvironment`. `sType` calculates the sType of the R expression `x`. `sTypeq` quotes its argument `x` before calling `sType`. SType inference of function definitions relies on a typed stack of formal arguments of `getSTypeFromFormalsStack` and `setSTypeOnFormalsStack` get or set the sType of a formal argument `x` and a `formalsStack`, respectively.

## Usage

```
sType(x, typeEnvir = rgpSTypeEnvironment, returnNullOnFailure = FALSE)

configureSTypeInference(constantSTypeFunction = NA)

calculateSTypeRecursive(x, typeEnvir = rgpSTypeEnvironment,
  formalsStack = list(), returnNullOnFailure = FALSE)
```

```
sTypeq(x, typeEnvir = rgpSTypeEnvironment, returnNullOnFailure = FALSE)

getSTypeFromFormalsStack(x, formalsStack)

setSTypeOnFormalsStack(x, value, formalsStack)

hasStype(x)

x %::% value
```

## Arguments

| | |
|---|---|
| `x` | The object to operate on. |
| `value` | An sType. |
| `typeEnvir` | The type environment, containing user-supplied sTypes of building blocks. |
| `formalsStack` | A stack of formal arguments with their sTypes. |
| `returnNullOnFailure` | Return NULL on failure instead of stopping, defaults to FALSE. |
| `constantSTypeFunction` | A function of one parameter to be used to calculate constant types. If set to `NA` (the default), types of constants are named after the constant's R class. |

## Details

The function `configureSTypeInference` is used to configure the type inference engine for special needs.

## See Also

sTypeConstructors

---

`subDataFrame` *Select a continuous subframe of a data frame*

---

## Description

Return a continuous subframe of the data frame `x` containing `size` * `nrow(x)` rows from the start, center, or end.

## Usage

```
subDataFrame(x, size = 1, pos = "START")
```

### Arguments

| | |
|---|---|
| `x` | The data frame to get a subframe from. |
| `size` | The size ratio of the subframe. Must be between 0 and 1. |
| `pos` | The position to take the subframe from. Must be `"START"`, `"CENTER"`, or `"END"`. |

### Value

A subframe of `x`.

---

## `subexpressions` *Functions for decomposing and recombining R expressions*

---

### Description

`subexpressions` returns a list of all subexpressions (subtrees) of an expression `expr`.

### Usage

```
subexpressions(expr)
```

### Arguments

| | |
|---|---|
| `expr` | An R expression. |

### Value

The decomposed or recombined expression.

---

## `summary.geneticProgrammingResult` *Summary reports of genetic programming run result objects*

---

### Description

Create a summary report of a genetic programming result object as returned by `geneticProgramming` or `symbolicRegression`, for example.

### Usage

```
## S3 method for class 'geneticProgrammingResult'
summary(object, reportFitness = FALSE,
  orderByFitness = TRUE, ...)
```

## Arguments

| | |
|---|---|
| `object` | The genetic programming run result object to report on. |
| `reportFitness` | Whether to report detailed fitness values of each individual in the result population. Note that calculating fitness values may take a long time. Defaults to `FALSE`. Either way, basic fitness values for each individual is reported. |
| `orderByFitness` | Whether the report of the result population should be ordered by fitness. This does not have an effect if `reportFitness` is set to `FALSE`. Defaults to `TRUE`. |
| `...` | Ignored in this summary function. |

## See Also

`geneticProgramming`, `symbolicRegression`

---

`symbolicRegression` *Symbolic regression via untyped standard genetic programming*

---

## Description

Perform symbolic regression via untyped genetic programming. The regression task is specified as a `formula`. Only simple formulas without interactions are supported. The result of the symbolic regression run is a symbolic regression model containing an untyped GP population of model functions.

## Usage

```
symbolicRegression(formula, data, stopCondition = makeTimeStopCondition(5),
  population = NULL, populationSize = 100, eliteSize = ceiling(0.1 *
  populationSize), elite = list(), extinctionPrevention = FALSE,
  archive = FALSE, individualSizeLimit = 64,
  penalizeGenotypeConstantIndividuals = FALSE, subSamplingShare = 1,
  functionSet = mathFunctionSet, constantSet = numericConstantSet,
  crossoverFunction = NULL, mutationFunction = NULL,
  restartCondition = makeEmptyRestartCondition(),
  restartStrategy = makeLocalRestartStrategy(),
  searchHeuristic = makeAgeFitnessComplexityParetoGpSearchHeuristic(),
  breedingFitness = function(individual) TRUE, breedingTries = 50,
  errorMeasure = rmse, progressMonitor = NULL, envir = parent.frame(),
  verbose = TRUE)
```

## Arguments

| | |
|---|---|
| `formula` | A `formula` describing the regression task. Only simple formulas of the form `response ~ variable1 + ... + variableN` are supported at this point in time. |
| `data` | A `data.frame` containing training data for the symbolic regression run. The variables in `formula` must match column names in this data frame. |

| | |
|---|---|
| `stopCondition` | The stop condition for the evolution main loop. See makeStepsStopCondition for details. |
| `population` | The GP population to start the run with. If this parameter is missing, a new GP population of size `populationSize` is created through random growth. |
| `populationSize` | The number of individuals if a population is to be created. |
| `eliteSize` | The number of elite individuals to keep. Defaults to `ceiling(0.1 * populationSize)`. |
| `elite` | The elite list, must be alist of individuals sorted in ascending order by their first fitness component. |
| `extinctionPrevention` | When set to `TRUE`, the initialization and selection steps will try to prevent duplicate individuals from occurring in the population. Defaults to `FALSE`, as this operation might be expensive with larger population sizes. |
| `archive` | If set to `TRUE`, all GP individuals evaluated are stored in an archive list `archiveList` that is returned as part of the result of this function. |
| `individualSizeLimit` | Individuals with a number of tree nodes that exceeds this size limit will get a fitness of `Inf`. |
| `penalizeGenotypeConstantIndividuals` | Individuals that do not contain any input variables will get a fitness of `Inf`. |
| `subSamplingShare` | The share of fitness cases $s$ sampled for evaluation with each function evaluation. $0 < s \leq 1$ must hold, defaults to `1.0`. |
| `functionSet` | The function set. |
| `constantSet` | The set of constant factory functions. |
| `crossoverFunction` | The crossover function. |
| `mutationFunction` | The mutation function. |
| `restartCondition` | The restart condition for the evolution main loop. See makeEmptyRestartCondition for details. |
| `restartStrategy` | The strategy for doing restarts. See makeLocalRestartStrategy for details. |
| `searchHeuristic` | The search-heuristic (i.e. optimization algorithm) to use in the search of solutions. See the documentation for `searchHeuristics` for available algorithms. |
| `breedingFitness` | A "breeding" function. This function is applied after every stochastic operation *Op* that creates or modifies an individal (typically, *Op* is a initialization, mutation, or crossover operation). If the breeding function returns `TRUE` on the |

| | |
|---|---|
| | given individual, *Op* is considered a success. If the breeding function returns `FALSE`, *Op* is retried a maximum of `breedingTries` times. If this maximum number of retries is exceeded, the result of the last try is considered as the result of *Op*. In the case the breeding function returns a numeric value, the breeding is repeated `breedingTries` times and the individual with the lowest breeding fitness is considered the result of *Op*. |
| `breedingTries` | In case of a boolean `breedingFitness` function, the maximum number of retries. In case of a numerical `breedingFitness` function, the number of breeding steps. Also see the documentation for the `breedingFitness` parameter. Defaults to `50`. |
| `errorMeasure` | A function to use as an error measure, defaults to RMSE. |
| `progressMonitor` | A function of signature `function(population, fitnessValues, fitnessFunction, stepNumber, e` to be called with each evolution step. |
| `envir` | The R environment to evaluate individuals in, defaults to `parent.frame()`. |
| `verbose` | Whether to print progress messages. |

## Value

An symbolic regression model that contains an untyped GP population.

## See Also

`predict.symbolicRegressionModel`, `geneticProgramming`

---

## `tabulateFunction` *Tabulate an n-ary function*

---

### Description

Creates a data frame of values for the n-ary function `f` at the sample locations given in `...`.

### Usage

```
tabulateFunction(f, ...)
```

### Arguments

| | |
|---|---|
| `f` | The function to tabulate. |
| `...` | For each dimension, a vector of sample points to calculate `f` at. |

### Value

A data frame of function values of `f`.

## toName *Functions for handling R symbols / names*

### Description

toName converts a character string x to an R symbol / name, while copying all attributes iff copyAttributes is TRUE. In the case that x is not a character string, a copy of the object is returned as-is. extractLeafSymbols returns the set of symbols (names) at the leafs of an expression expr. The symbols are returned as character strings.

### Usage

```
toName(x, copyAttributes = TRUE)

extractLeafSymbols(expr)
```

### Arguments

| | |
|---|---|
| x | The object to operate on. |
| expr | An R expression. |
| copyAttributes | Whether to copy all attributes of x to the result object. |

### Value

The result.

# Index