`rrapply`: revisiting R-base `rapply`

Joris Chau

November 7, 2020

Abstract

The minimal `rrapply`-package contains a single function `rrapply()`, providing an extended implementation of R-base's `rapply()` function. Base `rapply()` applies a function `f` to all elements of a list recursively. The `rrapply()` function extends base `rapply()` by including a condition or predicate function for the application of `f` and the option to prune or aggregate list elements from the result. In addition, special arguments `.xname`, `.xpos`, `.xparents` and `.xsiblings` can be used inside the `f` and `condition` functions to access the name, location, parents and siblings in the nested list of the list element under evaluation. `rrapply()` builds upon `rapply()`'s native C implementation and for this reason requires no external R-package dependencies.

1 Quick review of `rapply()`

The dataset `renewable_energy_by_country` included in the `rrapply`-package lists the share of renewable energy as a percentage in the total energy consumption per country in 2016. The dataset is publicly available at the United Nations Open SDG Data Hub (UNSD-SDG07). The 249 countries and areas are structured as a nested list based on their geographical location according to the United Nations M49 standard (UNSD-M49). The numeric values listed for each country are percentages, if no data is available the country's value is `NA`.

```
> library(rrapply)
> data("renewable_energy_by_country")
> ## display list structure (only first two elements of each node)
> str(renewable_energy_by_country, list.len = 2, give.attr = FALSE)

List of 1
 $ World:List of 6
  ..$ Africa    :List of 2
  .. ..$ Northern Africa   :List of 7
  .. .. ..$ Algeria       : num 0.08
  .. .. ..$ Egypt         : num 5.69
  .. .. .. [list output truncated]
```

```
  .. ..$ Sub-Saharan Africa:List of 4
  .. .. ..$ Eastern Africa :List of 22
  .. .. .. ..$ British Indian Ocean Territory: logi NA
  .. .. .. ..$ Burundi                       : num 89.2
  .. .. .. .. [list output truncated]
  .. .. ..$ Middle Africa  :List of 9
  .. .. .. ..$ Angola                         : num 54.6
  .. .. .. ..$ Cameroon                       : num 78.1
  .. .. .. .. [list output truncated]
  .. .. .. [list output truncated]
  ..$ Americas  :List of 2
  .. ..$ Latin America and the Caribbean:List of 3
  .. .. ..$ Caribbean      :List of 28
  .. .. .. ..$ Anguilla                       : num 0.11
  .. .. .. ..$ Antigua and Barbuda            : num 0
  .. .. .. .. [list output truncated]
  .. .. ..$ Central America:List of 8
  .. .. .. ..$ Belize     : num 30.3
  .. .. .. ..$ Costa Rica : num 37.2
  .. .. .. .. [list output truncated]
  .. .. .. [list output truncated]
  .. ..$ Northern America              :List of 5
  .. .. ..$ Bermuda                  : num 2.11
  .. .. ..$ Canada                   : num 21.6
  .. .. .. [list output truncated]
  .. [list output truncated]
```

For convenience, we subset only the values for countries and areas in `Oceania`,

```
> renewable_oceania <- renewable_energy_by_country[["World"]]["Oceania"]
> str(renewable_oceania, list.len = 3, give.attr = FALSE)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..$ Australia                      : num 9.32
  .. ..$ Christmas Island               : logi NA
  .. ..$ Cocos (Keeling) Islands        : logi NA
  .. .. [list output truncated]
```

```
  ..$ Melanesia                :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia               :List of 8
  .. ..$ Guam                                 : num 3.03
  .. ..$ Kiribati                             : num 45.4
  .. ..$ Marshall Islands                     : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

Using base `rapply()`, we can apply a function `f` to each leaf element or leaf elements of a particular class or type. By a leaf element, we refer to any element of the list which is not itself list-like, in this case the numeric country percentages. For instance, we can replace all `NA`'s by zeros using an `ifelse` statement in the `f` function,

```
> na_zero_oceania_unlist <- rapply(
    renewable_oceania,
    f = function(x) ifelse(is.na(x), 0, x)
  )
> head(na_zero_oceania_unlist)
```

```
                        Oceania.Australia and New Zealand.Australia
                                                               9.32
                  Oceania.Australia and New Zealand.Christmas Island
                                                               0.00
           Oceania.Australia and New Zealand.Cocos (Keeling) Islands
                                                               0.00
Oceania.Australia and New Zealand.Heard Island and McDonald Islands
                                                               0.00
                      Oceania.Australia and New Zealand.New Zealand
                                                              32.76
                    Oceania.Australia and New Zealand.Norfolk Island
                                                               0.00
```

By default, the result is returned *unlisted*. The original list structure can be preserved via the arguments `how = "replace"` or `how = "list"`. Conceptually, `how = "replace"` makes a complete copy of the input list and recursively replaces the leaf elements with a class in `classes` by the

result of applying `f`. `how = "list"` recursively makes copies of the list-like elements of the input list, replacing leaf elements with a class in `classes` by the result of applying `f`, and replacing any other leaf elements by the value of `deflt`. `how = "unlist"` calls `unlist()` with argument `recursive = TRUE` on the initial result obtained by `how = "list"`, thus allowing the use of the `deflt` argument.

By making use of the fact that the `NA`'s are of `logical` type and the non-`NA`'s are of `numeric` type, another way of replacing `NA`'s by zeros is via the `classes` argument:

```
> na_zero_oceania_replace <- rapply(
    renewable_oceania,
    f = function(x) 0,
    classes = "logical",
    how = "replace"
  )
> str(na_zero_oceania_replace, list.len = 3, give.attr = FALSE)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..$ Australia                       : num 9.32
  .. ..$ Christmas Island                : num 0
  .. ..$ Cocos (Keeling) Islands         : num 0
  .. .. [list output truncated]
  ..$ Melanesia                :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia               :List of 8
  .. ..$ Guam                                : num 3.03
  .. ..$ Kiribati                            : num 45.4
  .. ..$ Marshall Islands                    : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

Or, by combining the `classes` and `deflt` arguments together with `how = "list"` or `how = "unlist"`,

```
> na_zero_oceania_list <- rapply(
    renewable_oceania,
```

```
    f = function(x) x,
    classes = "numeric",
    deflt = 0,
    how = "list"
  )
> str(na_zero_oceania_list, list.len = 3, give.attr = FALSE)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..$ Australia                       : num 9.32
  .. ..$ Christmas Island                : num 0
  .. ..$ Cocos (Keeling) Islands         : num 0
  .. .. [list output truncated]
  ..$ Melanesia               :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia              :List of 8
  .. ..$ Guam                                : num 3.03
  .. ..$ Kiribati                            : num 45.4
  .. ..$ Marshall Islands                    : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

Each list element in `renewable_energy_by_country` contains an `"M49-code"` attribute with the "UN Standard Country or Area Codes for Statistical Use (Series M, No. 49)". In order to keep this attribute when replacing `NA`'s by zeros, we could modify the above call with `how = "replace"` to,

```
> na_zero_oceania_replace_attr <- rapply(
    renewable_oceania,
    f = function(x) replace(x, is.na(x), 0),
    how = "replace"
  )
> str(na_zero_oceania_replace_attr, list.len = 2)

List of 1
 $ Oceania:List of 4
```

```
  ..$ Australia and New Zealand:List of 6
  .. ..$ Australia                       : num 9.32
  .. .. ..- attr(*, "M49-code")= chr "036"
  .. ..$ Christmas Island                : num 0
  .. .. ..- attr(*, "M49-code")= chr "162"
  .. .. [list output truncated]
  .. ..- attr(*, "M49-code")= chr "053"
  ..$ Melanesia                :List of 5
  .. ..$ Fiji            : num 24.4
  .. .. ..- attr(*, "M49-code")= chr "242"
  .. ..$ New Caledonia   : num 4.03
  .. .. ..- attr(*, "M49-code")= chr "540"
  .. .. [list output truncated]
  .. ..- attr(*, "M49-code")= chr "054"
  .. [list output truncated]
  ..- attr(*, "M49-code")= chr "009"
```

With `how = "list"`, intermediate list attributes –excluding the leaf elements– are in general not preserved. For this reason, it is probably best to use `how = "replace"` whenever possible if list attributes are present and must be preserved.

```
> na_zero_oceania_list_attr <- rapply(
    renewable_oceania,
    f = function(x) replace(x, is.na(x), 0),
    how = "list"
  )
> ## this preserves all list attributes
> str(na_zero_oceania_replace_attr, max.level = 2)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..- attr(*, "M49-code")= chr "053"
  ..$ Melanesia                :List of 5
  .. ..- attr(*, "M49-code")= chr "054"
  ..$ Micronesia               :List of 8
  .. ..- attr(*, "M49-code")= chr "057"
  ..$ Polynesia                :List of 10
  .. ..- attr(*, "M49-code")= chr "061"
```

```
  ..- attr(*, "M49-code")= chr "009"
```

```
> ## this does not preserves all attributes!
> str(na_zero_oceania_list_attr, max.level = 2)
```

```
List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  ..$ Melanesia                :List of 5
  ..$ Micronesia               :List of 8
  ..$ Polynesia                :List of 10
```

2 When to use `rrapply()`

2.1 List pruning and unnesting

With base `rapply()` there is no convenient way to prune or filter leaf elements from the input list. Using the `deflt` argument, we could set all leaf elements that are not subject to application of `f` to e.g. `NA` or `NULL`, but we cannot drop these leaf elements altogether from the resulting list.

The `rrapply()` function adds an option to set the `how` argument to `how = "prune"`, in which case all leaf elements that are not subject to application of `f` are pruned from the list. The original list structure is retained, similar to the non-pruned options `how = "replace"` or `how = "list"`.

Using `how = "prune"`, we can drop all `NA` elements while preserving the original list structure:

```
> na_drop_oceania_list <- rrapply(
    renewable_oceania,
    f = function(x) x,
    classes = "numeric",
    how = "prune"
  )
> str(na_drop_oceania_list, list.len = 3, give.attr = FALSE)
```

```
List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 2
  .. ..$ Australia  : num 9.32
  .. ..$ New Zealand: num 32.8
  ..$ Melanesia                :List of 5
  .. ..$ Fiji            : num 24.4
```

```
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia               :List of 7
  .. ..$ Guam                           : num 3.03
  .. ..$ Kiribati                       : num 45.4
  .. ..$ Marshall Islands               : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

Instead, we can set `how = "flatten"` to return a flattened unnested version of the pruned list. This is more efficient than first returning the pruned list with `how = "prune"` and unlisting or flattening the list in a subsequent step.

```
> na_drop_oceania_flat <- rrapply(
    renewable_oceania,
    f = function(x) x,
    classes = "numeric",
    how = "flatten"
  )
> str(na_drop_oceania_flat, list.len = 10, give.attr = FALSE)

List of 22
 $ Australia                     : num 9.32
 $ New Zealand                   : num 32.8
 $ Fiji                          : num 24.4
 $ New Caledonia                 : num 4.03
 $ Papua New Guinea              : num 50.3
 $ Solomon Islands               : num 65.7
 $ Vanuatu                       : num 33.7
 $ Guam                          : num 3.03
 $ Kiribati                      : num 45.4
 $ Marshall Islands              : num 11.8
  [list output truncated]
```

Another option is to set `how = "melt"` to return a melted data.frame of the pruned list similar in format to `reshape2::melt()` applied to a nested list. The rows of the melted data.frame contain the node paths of the elements in the pruned list. The `"value"` column is a list-column with the values of the terminal nodes analogous to the flattened list returned by `how = "flatten"`.

```
> na_drop_melt <- rrapply(
    renewable_oceania,
    f = function(x) x,
    classes = "numeric",
    how = "melt"
  )
> head(na_drop_melt)
```

```
       L1                        L2              L3 value
1 Oceania Australia and New Zealand       Australia  9.32
2 Oceania Australia and New Zealand     New Zealand 32.76
3 Oceania                Melanesia            Fiji 24.36
4 Oceania                Melanesia   New Caledonia  4.03
5 Oceania                Melanesia Papua New Guinea 50.34
6 Oceania                Melanesia  Solomon Islands 65.73
```

If no names are present in a sublist of the input list, `how = "melt"` replaces the names in the melted data.frame by the list element indices `"..1"`, `"..2"`, and so on:

```
> ## Remove all names at L2 (these arguments are explained in the following sections)
> oceania_unnamed <- rrapply(
    renewable_oceania,
    classes = "list",
    condition = function(x, .xname) .xname == "Oceania",
    f = unname
  )
> na_drop_melt2 <- rrapply(
    oceania_unnamed,
    f = function(x) x,
    classes = "numeric",
    how = "melt"
  )
> head(na_drop_melt2)
```

```
       L1  L2              L3 value
1 Oceania ..1       Australia  9.32
2 Oceania ..1     New Zealand 32.76
3 Oceania ..2            Fiji 24.36
4 Oceania ..2  New Caledonia  4.03
```

```
5 Oceania ..2 Papua New Guinea 50.34
6 Oceania ..2  Solomon Islands 65.73
```

2.2 Condition function

Base `rapply()` allows to apply `f` to leaf elements of certain types or classes via the `classes` argument, which might not always provide sufficient control to partition leaf elements. For this purpose, `rrapply()` includes an additional `condition` argument, which accepts any principal argument function to use as a condition or predicate to select leaf elements to which `f` is applied. Conceptually, the `f` function is applied to all leaf elements for which the `condition` function exactly evaluates to `TRUE` similar to the `isTRUE` function. If the `condition` function is missing, `f` is applied to all leaf elements. In combination with `how = "prune"`, the `condition` function provides a flexible way to select and filter elements from the nested list.

Using the `condition` argument, we can update the above function call to better reflect our purpose:

```
> na_drop_oceania_list2 <- rrapply(
    renewable_oceania,
    condition = function(x) !is.na(x),
    f = function(x) x,
    how = "prune"
  )
> str(na_drop_oceania_list2, list.len = 3, give.attr = FALSE)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 2
  .. ..$ Australia  : num 9.32
  .. ..$ New Zealand: num 32.8
  ..$ Melanesia               :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia              :List of 7
  .. ..$ Guam                           : num 3.03
  .. ..$ Kiribati                       : num 45.4
  .. ..$ Marshall Islands               : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

`rrapply()` allows the `f` argument to be missing, in which case no function is applied to the leaf elements. Using the `Negate` function, we can rewrite the above expression somewhat more concisely as,

```
> na_drop_oceania_list3 <- rrapply(
    renewable_oceania,
    condition = Negate(is.na),
    how = "prune"
  )
> str(na_drop_oceania_list3, list.len = 3, give.attr = FALSE)
```

```
List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 2
  .. ..$ Australia  : num 9.32
  .. ..$ New Zealand: num 32.8
  ..$ Melanesia                :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia               :List of 7
  .. ..$ Guam                           : num 3.03
  .. ..$ Kiribati                       : num 45.4
  .. ..$ Marshall Islands               : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

A more interesting example is to consider a `condition` that is not also replicable using the `classes` argument. For instance, we can filter all countries with a renewable energy share above 85 percent, or all countries with a renewable energy share of 0 percent:

```
> renewable_energy_above_85 <- rrapply(
    renewable_energy_by_country,
    condition = function(x) x > 85,
    how = "prune"
  )
> str(renewable_energy_above_85, give.attr = FALSE)
```

```
List of 1
 $ World:List of 1
```

```
  ..$ Africa:List of 1
  .. ..$ Sub-Saharan Africa:List of 3
  .. .. ..$ Eastern Africa:List of 7
  .. .. .. ..$ Burundi                    : num 89.2
  .. .. .. ..$ Ethiopia                   : num 91.9
  .. .. .. ..$ Rwanda                     : num 86
  .. .. .. ..$ Somalia                    : num 94.7
  .. .. .. ..$ Uganda                     : num 88.6
  .. .. .. ..$ United Republic of Tanzania: num 86.1
  .. .. .. ..$ Zambia                     : num 88.5
  .. .. ..$ Middle Africa :List of 2
  .. .. .. ..$ Chad                              : num 85.3
  .. .. .. ..$ Democratic Republic of the Congo: num 97
  .. .. ..$ Western Africa:List of 1
  .. .. .. ..$ Guinea-Bissau: num 86.5

> ## passing arguments to condition via ...
> renewable_energy_equal_0 <- rrapply(
    renewable_energy_by_country,
    condition = `==`,
    e2 = 0,
    how = "prune"
  )
> str(renewable_energy_equal_0, give.attr = FALSE)

List of 1
 $ World:List of 4
  ..$ Americas:List of 1
  .. ..$ Latin America and the Caribbean:List of 1
  .. .. ..$ Caribbean:List of 1
  .. .. .. ..$ Antigua and Barbuda: num 0
  ..$ Asia    :List of 1
  .. ..$ Western Asia:List of 4
  .. .. ..$ Bahrain: num 0
  .. .. ..$ Kuwait : num 0
  .. .. ..$ Oman   : num 0
  .. .. ..$ Qatar  : num 0
  ..$ Europe  :List of 2
```

```
  .. ..$ Northern Europe:List of 1
  .. .. ..$ Channel Islands:List of 1
  .. .. .. ..$ Guernsey: num 0
  .. ..$ Southern Europe:List of 1
  .. .. ..$ Gibraltar: num 0
  ..$ Oceania :List of 2
  .. ..$ Micronesia:List of 1
  .. .. ..$ Northern Mariana Islands: num 0
  .. ..$ Polynesia :List of 1
  .. .. ..$ Wallis and Futuna Islands: num 0
```

Note that the `NA` elements are not returned, as the `condition` does not evaluate to `TRUE` for `NA` values.

As the `condition` function is a generalization of the `classes` argument to have more flexible control of the predicate, it is also possible to use the `deflt` argument together with `how = "list"` or `how = "unlist"` to set a default value to all leaf elements for which the `condition` does not evaluate to `TRUE`:

```
> na_zero_oceania_list2 <- rrapply(
    renewable_oceania,
    condition = Negate(is.na),
    deflt = 0,
    how = "list"
  )
> str(na_zero_oceania_list2, list.len = 3, give.attr = FALSE)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..$ Australia                      : num 9.32
  .. ..$ Christmas Island               : num 0
  .. ..$ Cocos (Keeling) Islands        : num 0
  .. .. [list output truncated]
  ..$ Melanesia               :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
```

```
  ..$ Micronesia               :List of 8
  .. ..$ Guam                                : num 3.03
  .. ..$ Kiribati                            : num 45.4
  .. ..$ Marshall Islands                    : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

To be consistent with base `rapply()`, the `deflt` argument can still only be used together with `how = "list"` or `how = "unlist"`. With `how = "replace"`, we can replace `NA` values by zeros using the `f` function in the same way as before,

```
> na_zero_oceania_replace2 <- rrapply(
    renewable_oceania,
    condition = is.na,
    f = function(x) 0,
    how = "replace"
  )
> str(na_zero_oceania_replace2, list.len = 3, give.attr = FALSE)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..$ Australia                       : num 9.32
  .. ..$ Christmas Island                : num 0
  .. ..$ Cocos (Keeling) Islands         : num 0
  .. .. [list output truncated]
  ..$ Melanesia                :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia               :List of 8
  .. ..$ Guam                                : num 3.03
  .. ..$ Kiribati                            : num 45.4
  .. ..$ Marshall Islands                    : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

2.2.1 Using the ... argument

In base `rapply()`, the first argument to `f` always evaluates to the content of the leaf element to which `f` is applied. Any further arguments (besides the special arguments `.xname`, `.xpos`, `.xparents` and `.xsiblings` discussed below) that are independent of the node content are supplied via the dots `...` argument. Since `rrapply()` accepts a function in two of its arguments `f` and `condition`, any further arguments defined via the `dots` also need to be defined as function arguments in *both* the `f` and `condition` function (if existing), even if they are not used in the function itself.

To illustrate, consider the following example where we replace all `NA` elements by a value defined in a separate argument `newvalue`:

```
> ## this is not ok!
> tryCatch({
    rrapply(
      renewable_oceania,
      condition = is.na,
      f = function(x, newvalue) newvalue,
      newvalue = 0,
      how = "replace"
    )
  }, error = function(error) error$message)

[1] "2 arguments passed to 'is.na' which requires 1"

> ## this is ok
> na_zero_oceania_replace3 <- rrapply(
    renewable_oceania,
    condition = function(x, newvalue) is.na(x),
    f = function(x, newvalue) newvalue,
    newvalue = 0,
    how = "replace"
  )
> str(na_zero_oceania_replace3, list.len = 3, give.attr = FALSE)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..$ Australia                        : num 9.32
```

```
  .. ..$ Christmas Island                : num 0
  .. ..$ Cocos (Keeling) Islands         : num 0
  .. .. [list output truncated]
  ..$ Melanesia                :List of 5
  .. ..$ Fiji            : num 24.4
  .. ..$ New Caledonia   : num 4.03
  .. ..$ Papua New Guinea: num 50.3
  .. .. [list output truncated]
  ..$ Micronesia               :List of 8
  .. ..$ Guam                                : num 3.03
  .. ..$ Kiribati                            : num 45.4
  .. ..$ Marshall Islands                    : num 11.8
  .. .. [list output truncated]
  .. [list output truncated]
```

2.3 Special arguments `.xname`, `.xpos`, `.xparents` and `.xsiblings`

For illustration purposes, let us return all non-missing values in `renewable_oceania` as a non-nested flattened list:

```
> renewable_oceania_flat <- rrapply(
    renewable_oceania,
    condition = Negate(is.na),
    how = "flatten"
  )
> str(renewable_oceania_flat, list.len = 10, give.attr = FALSE)

List of 22
 $ Australia                    : num 9.32
 $ New Zealand                  : num 32.8
 $ Fiji                         : num 24.4
 $ New Caledonia                : num 4.03
 $ Papua New Guinea             : num 50.3
 $ Solomon Islands              : num 65.7
 $ Vanuatu                      : num 33.7
 $ Guam                         : num 3.03
 $ Kiribati                     : num 45.4
 $ Marshall Islands             : num 11.8
  [list output truncated]
```

Suppose that we wish to apply a function to each list element that relies on the name of the node. A possible way to achieve this using `mapply()` would be:

```
> renewable_oceania_flat_text <- mapply(
    FUN = function(name, value) sprintf("Renewable energy in %s: %.2f%%", name, value),
    name = names(renewable_oceania_flat),
    value = renewable_oceania_flat,
    SIMPLIFY = FALSE
  )
> str(renewable_oceania_flat_text, list.len = 10)

List of 22
 $ Australia                    : chr "Renewable energy in Australia: 9.32%"
 $ New Zealand                  : chr "Renewable energy in New Zealand: 32.76%"
 $ Fiji                         : chr "Renewable energy in Fiji: 24.36%"
 $ New Caledonia                : chr "Renewable energy in New Caledonia: 4.03%"
 $ Papua New Guinea             : chr "Renewable energy in Papua New Guinea: 50.34%"
 $ Solomon Islands              : chr "Renewable energy in Solomon Islands: 65.73%"
 $ Vanuatu                      : chr "Renewable energy in Vanuatu: 33.67%"
 $ Guam                         : chr "Renewable energy in Guam: 3.03%"
 $ Kiribati                     : chr "Renewable energy in Kiribati: 45.43%"
 $ Marshall Islands             : chr "Renewable energy in Marshall Islands: 11.75%"
  [list output truncated]
```

Remark. Note that the `purrr`-package also contains the convenience function `imap` for exactly this purpose.

In base `rapply()`, the `f` function only has access to the content of a leaf element through its principal argument, but there is no convenient way to access the list element its name or location from inside the `f` function. This makes `rapply()` impractical if we want to apply a function `f` that relies on e.g. the name of the leaf element as in the above example.

To address this and similar issues, `rrapply()` allows the use of a number of special arguments `.xname`, `.xpos`, `.xparents` and `.xsiblings` in addition to the principal argument in the `f` and `condition` functions. The `.xname` argument evaluates to the name of the leaf element. The `.xpos` argument evaluates to the position of the leaf element in the nested list structured as an integer vector. For instance, if `x = list(list("y", "z"))`, then an `.xpos` location of `c(1, 2)` corresponds to the leaf element `x[[1]][[2]]` or equivalently `x[[c(1, 2)]]`. The `.xparents` argument evaluates to a vector of all parent node names in the path to the list element. The names in `.xparents` are ordered by increasing depth in the nested list, with the final element always

matching to `.xname`. The `.xsiblings` argument evaluates to the parent list containing the current list element and all of its direct siblings.

The arguments `.xname`, `.xpos`, `.xparents` and `.xsiblings` need to be defined explicitly as function arguments in `f` and `condition` whenever they are used. Note that the principal function arguments of `f` and `condition` always evaluate to the content of the list element, for this reason the arguments `.xname`, `.xpos`, `.xparents` and `.xsiblings` should always be defined in addition to a principal function argument. It is allowed to use any combination of special arguments in `f` and `condition`.

Using the `.xname` argument, we can reproduce the `mapply()` example above also from a nested list as input:

```
> renewable_oceania_flat_text <- rrapply(
    renewable_oceania,
    f = function(x, .xname) sprintf("Renewable energy in %s: %.2f%%", .xname, x),
    condition = Negate(is.na),
    how = "flatten"
  )
> str(renewable_oceania_flat_text, list.len = 10)

List of 22
 $ Australia                    : chr "Renewable energy in Australia: 9.32%"
 $ New Zealand                  : chr "Renewable energy in New Zealand: 32.76%"
 $ Fiji                         : chr "Renewable energy in Fiji: 24.36%"
 $ New Caledonia                : chr "Renewable energy in New Caledonia: 4.03%"
 $ Papua New Guinea             : chr "Renewable energy in Papua New Guinea: 50.34%"
 $ Solomon Islands              : chr "Renewable energy in Solomon Islands: 65.73%"
 $ Vanuatu                      : chr "Renewable energy in Vanuatu: 33.67%"
 $ Guam                         : chr "Renewable energy in Guam: 3.03%"
 $ Kiribati                     : chr "Renewable energy in Kiribati: 45.43%"
 $ Marshall Islands             : chr "Renewable energy in Marshall Islands: 11.75%"
  [list output truncated]
```

Since these special arguments can also be used in the `condition` function, it is now possible to filter elements or apply a function only to a part of the list based on the node names or their positions.

As an example, let us extract the renewable energy shares of Belgium, the Netherlands and Luxembourg while preserving the nested structure of the filtered elements:

```
> renewable_benelux <- rrapply(
    renewable_energy_by_country,
    condition = function(x, .xname) .xname %in% c("Belgium", "Netherlands", "Luxembourg"),
    how = "prune"
  )
> str(renewable_benelux, give.attr = FALSE)

List of 1
 $ World:List of 1
  ..$ Europe:List of 1
  .. ..$ Western Europe:List of 3
  .. .. ..$ Belgium    : num 9.14
  .. .. ..$ Luxembourg : num 13.5
  .. .. ..$ Netherlands: num 5.78
```

Knowing that Europe is located under the node `renewable_energy_by_country[[c(1, 5)]]`, we can filter all European countries with a renewable energy share above 50 percent by using the `.xpos` argument,

```
> renewable_europe_above_50 <- rrapply(
    renewable_energy_by_country,
    condition = function(x, .xpos) identical(head(.xpos, 2), c(1L, 5L)) & x > 50,
    how = "prune"
  )
> str(renewable_europe_above_50, give.attr = FALSE)

List of 1
 $ World:List of 1
  ..$ Europe:List of 2
  .. ..$ Northern Europe:List of 3
  .. .. ..$ Iceland: num 78.1
  .. .. ..$ Norway : num 59.5
  .. .. ..$ Sweden : num 51.4
  .. ..$ Western Europe :List of 1
  .. .. ..$ Liechtenstein: num 62.9
```

This can be done more conveniently using the `.xparents` argument, as this does not require us to look up the location of Europe in the list beforehand,

```
> renewable_europe_above_50 <- rrapply(
    renewable_energy_by_country,
```

```
    condition = function(x, .xparents) "Europe" %in% .xparents & x > 50,
    how = "prune"
  )
```

Using the `.xpos` argument, we could also look up the location of a particular country in the nested list,

```
> (xpos_sweden <- rrapply(
    renewable_energy_by_country,
    condition = function(x, .xname) identical(.xname, "Sweden"),
    f = function(x, .xpos) .xpos,
    how = "flatten"
  ))

$Sweden
[1]  1  5  2 14

> ## sanity check
> renewable_energy_by_country[[xpos_sweden$Sweden]]

[1] 51.35
attr(,"M49-code")
[1] "752"
```

We could even use the `.xpos` argument to determine the maximum depth of the list or the length of the longest sublist,

```
> ## maximum depth
> depth_all <- rrapply(
    renewable_energy_by_country,
    f = function(x, .xpos) length(.xpos),
    how = "unlist"
  )
> max(depth_all)

[1] 5

> ## longest sublist length
> sublist_count <- rrapply(
    renewable_energy_by_country,
    f = function(x, .xpos) max(.xpos),
    how = "unlist"
```

```
  )
> max(sublist_count)

[1] 28
```

The `.xsiblings` argument allows us to evaluate the sibling nodes of the current list element under evaluation. An example usage is to look up all direct neighbors of a particular country in the nested list:

```
> ## look up sibling countries of Sweden in list
> siblings_sweden <- rrapply(
    renewable_energy_by_country,
    condition = function(x, .xsiblings) "Sweden" %in% names(.xsiblings),
    how = "flatten"
  )
> str(siblings_sweden, give.attr = FALSE)

List of 14
 $ Aland Islands                                    : logi NA
 $ Denmark                                          : num 33.1
 $ Estonia                                          : num 26.6
 $ Faroe Islands                                    : num 4.24
 $ Finland                                          : num 42
 $ Iceland                                          : num 78.1
 $ Ireland                                          : num 8.65
 $ Isle of Man                                      : num 4.3
 $ Latvia                                           : num 38.5
 $ Lithuania                                        : num 31.4
 $ Norway                                           : num 59.5
 $ Svalbard and Jan Mayen Islands                   : logi NA
 $ Sweden                                           : num 51.4
 $ United Kingdom of Great Britain and Northern Ireland: num 8.77
```

2.4 Avoid recursing into list nodes

By default, if `classes = "ANY"` both base `rapply()` and `rrapply()` recurse into any list-like element. Using `classes = "list"` in base `rapply()` has no effect as the function descends into any list node before evaluating the `classes` argument. In contrast, `rrapply()` does detect `classes = "list"`, in which case the `f` function is applied to list elements that satisfy the `condition` function. If the `condition` is not satisfied for a list element, `rrapply()` will recurse

further into the sublist, apply the `f` function to the nodes that satisfy `condition` and so on. The use of `classes = "list"` tells `rrapply()` to not descend into list objects by default. For this reason this behavior can only be triggered with the `classes` argument and *not* through the use of e.g. `condition = is.list`.

The choice `classes = "list"` is useful to calculate summary statistics across nodes or to look up the position of intermediate nodes in a nested list. To illustrate, we can return the mean and standard deviation of the renewable energy share in Europe as follows:

```
> rrapply(
    renewable_energy_by_country,
    classes = "list",
    condition = function(x, .xname) .xname == "Europe",
    f = function(x) list(
      mean = mean(unlist(x), na.rm = TRUE),
      sd = sd(unlist(x), na.rm = TRUE)
    ),
    how = "flatten"
  )

$Europe
$Europe$mean
[1] 22.36565


$Europe$sd
[1] 17.12639
```

Remark. Note that the principal `x` argument in the `f` function now evaluates to a list for the node satisfying `condition`. For this reason, we first `unlist` the sublist before passing it to `mean` and `sd`.

The same result could be obtained by defining a `condition` based on the `"M49-code"` attribute of the list element. This can be convenient to filter or summarize nodes in nested lists coming from XML (or HTML) files based on their attribute values.

```
> rrapply(
    renewable_energy_by_country,
    classes = "list",
    condition = function(x) attr(x, "M49-code") == "150",
    f = function(x) list(
```

```
      mean = mean(unlist(x), na.rm = TRUE),
      sd = sd(unlist(x), na.rm = TRUE)
    ),
    how = "flatten"
  )
```

```
$Europe
$Europe$mean
[1] 22.36565


$Europe$sd
[1] 17.12639
```

We can use the `.xpos` argument to apply the `f` function only at specific locations or depths in the nested list. For instance, we could return the mean renewable energy shares for each continent by observing that the `.xpos` vector of each continent has length (i.e. depth) 2:

```
> renewable_continent_summary <- rrapply(
    renewable_energy_by_country,
    classes = "list",
    condition = function(x, .xpos) length(.xpos) == 2,
    f = function(x) mean(unlist(x), na.rm = TRUE)
  )
> ## Antarctica has a missing value
> str(renewable_continent_summary, give.attr = FALSE)
```

```
List of 1
 $ World:List of 6
  ..$ Africa    : num 54.3
  ..$ Americas  : num 18.2
  ..$ Antarctica: logi NA
  ..$ Asia      : num 17.9
  ..$ Europe    : num 22.4
  ..$ Oceania   : num 17.8
```

If `classes = "list"`, the `f` function is only applied to the (non-terminal) list nodes. If `classes = "ANY"`, the `f` function is only applied to any (terminal) non-list node. To apply `f` to any terminal *and* non-terminal node of the nested list, we should combine `classes = c("list", "ANY")`. This is illustrated by searching both terminal and non-terminal nodes for the country or region with M49-code `"155"` (Western Europe):

```
> ## Filter country or region by M49-code
> rrapply(
    renewable_energy_by_country,
    classes = c("list", "ANY"),
    condition = function(x) attr(x, "M49-code") == "155",
    f = function(x, .xname) .xname,
    how = "unlist")

World.Europe.Western Europe
           "Western Europe"
```

2.5 Recursive list node modification

If `classes = "list"`, `rrapply()` applies the `f` function to any list element that satisfies the `condition` function, but will not recurse further into these list elements. This make it for instance impossible to recursively update the name of each list element in a nested list, as `rrapply()` stops recursing after updating the first list layer. For this purpose, set `classes = "list"` combined with `how = "recurse"`, in which case `rrapply()` recurses further into any *updated* list element after application of the `f` function (using `how = "replace"`). In this context, the `condition` argument is interpreted as a passing criterion: as long as the `condition` and `classes` arguments are satisfied, `rrapply()` will try to recurse further into any list-like element.

Using `how = "recurse"`, we can recursively replace all names in the nested list by their M49-codes:

```
> renewable_M49 <- rrapply(
    list(renewable_energy_by_country),
    classes = "list",
    f = function(x) {
      names(x) <- vapply(x, attr, character(1L), which = "M49-code")
      return(x)
    },
    how = "recurse"
  )
> str(renewable_M49[[1]], max.level = 3, list.len = 3, give.attr = FALSE)

List of 1
 $ 001:List of 6
  ..$ 002:List of 2
  .. ..$ 015:List of 7
  .. ..$ 202:List of 4
```

```
  ..$ 019:List of 2
  .. ..$ 419:List of 3
  .. ..$ 021:List of 5
  ..$ 010: logi NA
  .. [list output truncated]
```

Remark. Here we passed `list(renewable_energy_by_country)` to the call of `rrapply()` in order to start application of the `f` function at the level of the list `renewable_energy_by_country` itself, instead of starting at its list elements.

2.6 Miscellaneous

2.6.1 Unmelting data.frames to lists

The option `how = "melt"` returns a melted data.frame similar in format as `reshape2::melt()` applied to a nested list. Suppose that we have manipulated the data.frame, by e.g. filtering a number of rows, and wish to convert it back to a nested list in order to write it to e.g. a JSON- or XML-object. For this purpose, `rrapply()` includes the option `how = "unmelt"`, which performs the inverse operation of `how = "melt"`. No skeleton object is needed in this case (as required in e.g. base R's `relist()` function), only a data.frame in the same format as returned by `how = "melt"`. To illustrate, we can reconstruct a nested list from the data.frame `na_drop_melt` produced above as follows,

```
> ## melted data.frame
> head(na_drop_melt)
```

```
      L1                        L2              L3 value
1 Oceania Australia and New Zealand       Australia  9.32
2 Oceania Australia and New Zealand     New Zealand 32.76
3 Oceania                Melanesia             Fiji 24.36
4 Oceania                Melanesia    New Caledonia  4.03
5 Oceania                Melanesia Papua New Guinea 50.34
6 Oceania                Melanesia  Solomon Islands 65.73
```

```
> na_drop_unmelt <- rrapply(na_drop_melt, how = "unmelt")
> str(na_drop_unmelt, list.len = 3, give.attr = FALSE)
```

```
List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 2
  .. ..$ Australia  : num 9.32
```

```
.. ..$ New Zealand: num 32.8
..$ Melanesia               :List of 5
.. ..$ Fiji            : num 24.4
.. ..$ New Caledonia   : num 4.03
.. ..$ Papua New Guinea: num 50.3
.. .. [list output truncated]
..$ Micronesia              :List of 7
.. ..$ Guam                          : num 3.03
.. ..$ Kiribati                      : num 45.4
.. ..$ Marshall Islands              : num 11.8
.. .. [list output truncated]
.. [list output truncated]
```

Remark. `how = "unmelt"` is based on a greedy approach parsing data.frame rows as list elements starting from the top of the data.frame. That is, `rrapply()` continues collecting children nodes as long as the parent node name remains unchanged. If, for instance, we wish to create two separate nodes (on the same list layer) with the name `"Western Europe"`, these nodes should not be listed after one another in the melted data.frame as `rrapply()` will group all children under a single `"Western Europe"` list element.

Remark. Internally, `how = "unmelt"` reconstructs a nested list from the melted data.frame and subsequently applies the same conceptual framework as `how = "replace"`. Any other function arguments, such as `f` and `condition`, should be used in exactly the same way as one would use them with `how = "replace"` applied to a nested list object.

Remark. `how = "unmelt"` does (currently) not restore the attributes of intermediate list nodes and is therefore not an exact inverse of `how = "melt"`. Melting an already unmelted nested list will always produce exactly the same results:

```
> na_drop_remelt <- rrapply(na_drop_unmelt, how = "melt")
> identical(na_drop_melt, na_drop_remelt)

[1] TRUE
```

2.6.2 Data.frames as lists

If `classes = "ANY"`, `rrapply()` recurses into any list-like object equivalent to base `rapply()`. Since data.frames are list-like objects, the `f` function is applied to the individual columns instead of the data.frame object as a whole. To avoid this behavior, set `classes = "data.frame"`, in which case the `f` and `condition` functions are applied directly to the data.frame object itself and

not its columns. Note that with base `rapply()` using `classes = "data.frame"` has no effect as `rapply()` descends into the columns of any data.frame object before evaluating the `classes` argument.

```
> ## create a list of data.frames
> oceania_df <- list(
    Oceania = lapply(
      renewable_oceania[["Oceania"]],
      FUN = function(x) data.frame(value = unlist(x))
    )
  )
> ## this does not work!
> tryCatch({
    rrapply(
      oceania_df,
      f = function(x) subset(x, !is.na(value)), ## filter NA-rows of data.frame
      how = "replace"
    )
  }, error = function(error) error$message)

[1] "object 'value' not found"

> ## this does work
> rrapply(
    oceania_df,
    classes = "data.frame",
    f = function(x) subset(x, !is.na(value)),
    how = "replace"
  )

$Oceania
$Oceania$`Australia and New Zealand`
            value
Australia    9.32
New Zealand 32.76

$Oceania$Melanesia
                value
Fiji            24.36
```

```
New Caledonia     4.03
Papua New Guinea 50.34
Solomon Islands  65.73
Vanuatu          33.67

$Oceania$Micronesia
                                value
Guam                             3.03
Kiribati                        45.43
Marshall Islands                11.75
Micronesia (Federated States of)  1.64
Nauru                           31.44
Northern Mariana Islands         0.00
Palau                            0.02

$Oceania$Polynesia
                         value
American Samoa            1.00
Cook Islands              1.90
French Polynesia         11.06
Niue                     22.07
Samoa                    27.30
Tonga                     1.98
Tuvalu                   11.76
Wallis and Futuna Islands 0.00
```

Remark. Note that the same result can also be obtained using `classes = "list"` and checking that the list element under evaluation is a data.frame:

```
> rrapply(
    oceania_df,
    classes = "list",
    condition = is.data.frame,
    f = function(x) subset(x, !is.na(value)),
    how = "replace"
  )
```

2.6.3 List attributes

Base `rapply()` may produce different results when using `how = "replace"` or `how = "list"` when working with list attributes. The former preserves intermediate list attributes whereas the latter does not. To avoid unexpected behavior, `rrapply()` always preserves intermediate list attributes when using `how = "replace"`, `how = "list"` or `how = "prune"`. Note that if we set `how = "flatten"`, `how = "melt"` or `how = "unlist"` intermediate list attributes cannot be preserved as the result is no longer a nested list.

```
> ## how = "list" now preserves all list attributes
> na_drop_oceania_list_attr2 <- rrapply(
    renewable_oceania,
    f = function(x) replace(x, is.na(x), 0),
    how = "list"
  )
> str(na_drop_oceania_list_attr2, max.level = 2)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 6
  .. ..- attr(*, "M49-code")= chr "053"
  ..$ Melanesia                :List of 5
  .. ..- attr(*, "M49-code")= chr "054"
  ..$ Micronesia               :List of 8
  .. ..- attr(*, "M49-code")= chr "057"
  ..$ Polynesia                :List of 10
  .. ..- attr(*, "M49-code")= chr "061"
  ..- attr(*, "M49-code")= chr "009"

> ## how = "prune" also preserves list attributes
> na_drop_oceania_attr <- rrapply(
    renewable_oceania,
    condition = Negate(is.na),
    how = "prune"
  )
> str(na_drop_oceania_attr, max.level = 2)

List of 1
 $ Oceania:List of 4
  ..$ Australia and New Zealand:List of 2
```

```
  .. ..- attr(*, "M49-code")= chr "053"
  ..$ Melanesia                :List of 5
  .. ..- attr(*, "M49-code")= chr "054"
  ..$ Micronesia               :List of 7
  .. ..- attr(*, "M49-code")= chr "057"
  ..$ Polynesia                :List of 8
  .. ..- attr(*, "M49-code")= chr "061"
  ..- attr(*, "M49-code")= chr "009"
```

2.7 Using `rrapply()` on data.frames

The previous sections explained how to avoid recursing into list-like elements using `rrapply()`. However, it can also be useful to exploit the property that a data.frame is a list-like object and use base `rapply()` to apply a function `f` to data.frame columns of certain classes. For instance, it is straightforward to standardize all `numeric` columns in the `iris` dataset by their sample mean and standard deviation:

```
> iris_standard <- rapply(iris, f = scale, classes = "numeric", how = "replace")
> head(iris_standard)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1   -0.8976739  1.01560199    -1.335752   -1.311052  setosa
2   -1.1392005 -0.13153881    -1.335752   -1.311052  setosa
3   -1.3807271  0.32731751    -1.392399   -1.311052  setosa
4   -1.5014904  0.09788935    -1.279104   -1.311052  setosa
5   -1.0184372  1.24503015    -1.335752   -1.311052  setosa
6   -0.5353840  1.93331463    -1.165809   -1.048667  setosa
```

Using the `condition` argument in `rrapply()`, we obtain more flexible control in selecting the columns to which `f` is applied. For instance, it is now straightforward to apply the `f` function only to the `Sepal` columns using the `.xname` argument:

```
> iris_standard_sepal <- rrapply(
    iris,
    condition = function(x, .xname) grepl("Sepal", .xname),
    f = scale
  )
> head(iris_standard_sepal)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1   -0.8976739  1.01560199          1.4         0.2  setosa
```

```
2   -1.1392005 -0.13153881          1.4         0.2  setosa
3   -1.3807271  0.32731751          1.3         0.2  setosa
4   -1.5014904  0.09788935          1.5         0.2  setosa
5   -1.0184372  1.24503015          1.4         0.2  setosa
6   -0.5353840  1.93331463          1.7         0.4  setosa
```

Instead of *mutating* columns, we can also *transmute* columns (referencing to the semantics of the `dplyr`-package) keeping only the columns to which `f` is applied by setting `how = "prune"`:

```
> iris_standard_transmute <- rrapply(
    iris,
    f = scale,
    classes = "numeric",
    how = "prune"
  )
> head(iris_standard_transmute)

  Sepal.Length Sepal.Width Petal.Length Petal.Width
1   -0.8976739  1.01560199    -1.335752   -1.311052
2   -1.1392005 -0.13153881    -1.335752   -1.311052
3   -1.3807271  0.32731751    -1.392399   -1.311052
4   -1.5014904  0.09788935    -1.279104   -1.311052
5   -1.0184372  1.24503015    -1.335752   -1.311052
6   -0.5353840  1.93331463    -1.165809   -1.048667
```

In order to *summarize* a set of selected columns, use `how = "flatten"` instead of `how = "prune"`, as the latter preserves list attributes –including data.frame dimensions– which should not be kept.

```
> ## summarize columns with how = "flatten"
> iris_standard_summarize <- rrapply(
    iris,
    f = summary,
    classes = "numeric",
    how = "flatten"
  )
> iris_standard_summarize

$Sepal.Length
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300   5.100   5.800   5.843   6.400   7.900
```

```
$Sepal.Width
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   2.800   3.000   3.057   3.300   4.400

$Petal.Length
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.600   4.350   3.758   5.100   6.900

$Petal.Width
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.100   0.300   1.300   1.199   1.800   2.500
```

3 Using `rrapply()` on expressions

Unevaluated R code is parsed or captured as a set of *expressions*, which is a collective term used to refer to any of the following types of objects: scalar constants e.g. `TRUE` or `1`, symbols e.g. `quote(x)`, call objects e.g. `quote(x <- 1)`, expression vectors e.g. `expression(a <- 1, 2 * b)` or pairlists e.g. `formals(seq.default)`. Call objects and expression vectors are hierarchically structured objects, i.e. *abstract syntax trees*, that can be decomposed into symbols and scalar constants as atomic building blocks.

Call objects and expression vectors generally behave as nested lists, e.g. subsetting a call object works the same as subsetting a nested list. To illustrate, we can retrieve the abstract syntax tree of a call object by recursing through the object in the same way as for a nested list:

```
> ## recurse through call as nested list
> ast <- function(expr) {
    lapply(expr, function(x) {
      if(is.call(x) || is.expression(x)) {
        ast(x)
      } else {
        x
      }
    })
  }
> ## decompose call object
> str(ast(quote(y <- x <- 1 + TRUE)))
```

```
List of 3
 $ : symbol <-
 $ : symbol y
 $ :List of 3
  ..$ : symbol <-
  ..$ : symbol x
  ..$ :List of 3
  .. ..$ : symbol +
  .. ..$ : num 1
  .. ..$ : logi TRUE
```

Given this information, we might be inclined to assume that base `rapply()` also recurses through call objects and expression vectors in the same way as for nested lists, but this is unfortunately not the case. To be precise, `rapply()` does accept expression vectors as input for R >= 3.6, but effectively treats them as flat lists of call objects similar to `lapply()`. Call objects are not accepted by `rapply()` (for any R version) and return an error.

```
> ## rapply on an expression vector (ok for R >= 3.6)
> tryCatch({
    rapply(expression(y <- x <- 1, f(g(2 * pi))), f = identity)
  }, error = function(error) error$message)

[[1]]
y <- x <- 1


[[2]]
f(g(2 * pi))
```

```
> ## rapply on a call object (not ok!)
> tryCatch({
    rapply(quote(y <- x <- 1), f = identity)
  }, error = function(error) error$message)

[1] "'object' must be a list or expression"
```

Starting from version 1.2.0 `rrapply()` also supports recursion of call objects and expression vectors, which are treated as nested lists based on their internal abstract syntax trees. As such, all functionality described in the previous sections extends directly to call objects and expression vectors.

3.1 Structuring the result

When applying `rrapply()` (or base `rapply()`) to nested lists the difference between `how = "replace"` and `how = "list"` is relatively minor. Both choices return a nested list, but only `how = "list"` replaces elements not subject to `f` by the argument `deflt`. For call objects and expression objects, the difference is more important as `how = "replace"` always maintains the type of the object after application of `rrapply()`, whereas `how = "list"` returns the object formatted as a nested list.

With `how = "replace"`, we can for instance directly update the abstract syntax tree of a call object:

```
> call_old <- quote(y <- x <- 1 + TRUE)
> str(call_old)

 language y <- x <- 1 + TRUE

> ## update call object
> call_new <- rrapply(
    call_old,
    classes = "logical",
    f = as.numeric,
    how = "replace"
  )
> str(call_new)

 language y <- x <- 1 + 1
```

Using `how = "list"`, we can update the abstract syntax tree and return it as a nested list:

```
> ## update and decompose call object
> call_ast <- rrapply(
    call_old,
    f = function(x) ifelse(is.logical(x), as.numeric(x), x),
    how = "list"
  )
> str(call_ast)

List of 3
 $ : symbol <-
 $ : symbol y
 $ :List of 3
  ..$ : symbol <-
```

```
  ..$ : symbol x
  ..$ :List of 3
  .. ..$ : symbol +
  .. ..$ : num 1
  .. ..$ : num 1
```

Remark. Note that in the second function call we did not use `classes = "logical"` to avoid that all list elements that are not of class `"logical"` are replaced by the `deflt` argument (`NULL`).

The choices `how = "prune"`, `how = "flatten"` and `how = "melt"` return the pruned abstract syntax tree as: a nested list, a flattened list and a melted data.frame respectively. This is identical to application of `rrapply()` to the abstract syntax tree formatted as a nested list. To illustrate, let us return all names in the abstract syntax tree of an expression vector that are not part of base R.

```
> ## example expression
> expr <- expression(y <- x <- 1, f(g(2 * pi)))
> ## helper function
> is_new_name <- function(x) !exists(as.character(x), envir = baseenv())

> ## prune and decompose expression
> expr_prune <- rrapply(
    expr,
    classes = "name",
    condition = is_new_name,
    how = "prune"
  )
> str(expr_prune)

List of 2
 $ :List of 2
  ..$ : symbol y
  ..$ :List of 1
  .. ..$ : symbol x
 $ :List of 2
  ..$ : symbol f
  ..$ :List of 1
  .. ..$ : symbol g

> ## prune and flatten expression
> expr_flatten <- rrapply(
```

```
    expr,
    classes = "name",
    condition = is_new_name,
    how = "flatten"
  )
> str(expr_flatten)

List of 4
 $ : symbol y
 $ : symbol x
 $ : symbol f
 $ : symbol g

> ## prune and melt expression
> expr_melt <- rrapply(
    expr,
    classes = "name",
    condition = is_new_name,
    f = as.character,
    how = "melt"
  )
> expr_melt

   L1  L2   L3 value
1 ..1 ..2 <NA>     y
2 ..1 ..3  ..2     x
3 ..2 ..1 <NA>     f
4 ..2 ..2  ..1     g
```

3.2 Avoid recursing into expression nodes

If `classes = "ANY"` (the default), `rrapply()` recurses into any list-like element, which for expression objects are: call objects, expression vectors and pairlists. To avoid recursing into list elements of a nested list, we can use `classes = "list"`. Analogously, to avoid recursing into list-like elements of the abstract syntax tree, we should use `classes = "language"`, `classes = "expression"`, `classes = "pairlist"` or any combination thereof. If the `condition` and `classes` arguments are not satisfied for a given list-like element, `rrapply()` will recurse further into the object, apply the `f` function to the nodes that satisfy `condition` and `classes` and so on. Note that this behavior can only be triggered with the `classes` argument and *not* through the use of e.g.

`condition = is.call`.

To illustrate, we extract all most deeply nested call objects of the example expression above and return these as a flat list:

```
> ## extract all terminal call nodes of AST
> rrapply(
    expr,
    classes = "language",
    condition = function(x) !any(sapply(x, is.call)),
    how = "flatten"
  )

[[1]]
x <- 1


[[2]]
2 * pi
```

For additional details and worked out code examples, also visit the *Articles* section at the package website: https://jorischau.github.io/rrapply/.