`rrapply`: revisiting R-base `rapply`

Joris Chau

May 6, 2020

Abstract

The `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 list elements from the result. In addition, special symbols `.xname` and `.xpos` can be used inside the `f` and `condition` functions to access the name and location in the nested list of the list element under evaluation. The `rrapply` function is implemented using R's native C API 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

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]
```

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 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 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 symbols `.xname` and `.xpos`

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 and 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 issue, `rrapply` allows the use of two special symbols `.xname` and `.xpos` inside the `f` and `condition` functions. The `.xname` symbol evaluates to the name of the leaf element. The `.xpos` symbol 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 names `.xname` and `.xpos` do not need to be included as function arguments in `f` and `condition`, and can be thought of as pre-defined variables in the current function environment.

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

```
> renewable_oceania_flat_text <- rrapply(
    renewable_oceania,
    f = function(x) 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 the `.xname` and `.xpos` variables 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 %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` symbol,

```
> renewable_europe_above_50 <- rrapply(
    renewable_energy_by_country,
    condition = function(x) 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
```

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) identical(.xname, "Sweden"),
    f = function(x) .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` symbol 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) length(.xpos),
    how = "unlist"
  )
> max(depth_all)
```

```
[1] 5
```

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

```
[1] 28
```

Although not recommended, it is possible to override the values of `.xname` and `.xpos` by defining these variables as function arguments in `f` and `condition` and providing values via the `...` argument:

```
> ## override .xname
> rrapply(renewable_energy_by_country,
          condition = function(x, .xname) all(.xpos == 1), ## returns first element only
          f = function(x, .xname) paste(".xname is", .xname),
          .xname = "not the node name!",
          how = "flatten"
  )
```

```
$Algeria
[1] ".xname is not the node name!"
```

Remark. Defining new values for `.xname` or `.xpos` in the parent environment in which `rrapply` is called does not override the values for `.xname` or `.xpos` in the `f` or `condition` functions:

```
> ## this does not override .xname
> .xname <- "not the node name!"
> rrapply(renewable_energy_by_country,
        condition = function(x) all(.xpos == 1), ## returns first element only
        f = function(x) paste(".xname is", .xname),
        how = "flatten"
  )

$Algeria
[1] ".xname is Algeria"

> ## variable remains unaltered
> .xname

[1] "not the node name!"
```

2.4 Miscellanous

2.4.1 Data.frames as lists

Base `rapply` recurses into all list-like objects. Since data.frames are list-like objects, the `f` function always descends into the individual columns of a data.frame. It might occur that we wish to apply `f` to a data.fame object as a whole, instead of its individual columns, which is not possible with `rapply`. For this purpose, `rrapply` includes an additional argument `dfAsList`. If `dfAsList = TRUE`, `rrapply` behaves in the same way as `rapply` by recursing into the individual columns of a data.frame. If `dfAsList = FALSE`, the `f` and `condition` functions are applied directly to the data.frame object itself and not its columns.

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

[1] "object 'value' not found"

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

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


$Oceania$Melanesia
                             Name value
Fiji                         Fiji 24.36
New Caledonia       New Caledonia  4.03
Papua New Guinea Papua New Guinea 50.34
Solomon Islands   Solomon Islands 65.73
Vanuatu                   Vanuatu 33.67


$Oceania$Micronesia
                                                                Name
Guam                                                            Guam
Kiribati                                                    Kiribati
Marshall Islands                                    Marshall Islands
Micronesia (Federated States of) Micronesia (Federated States of)
Nauru                                                          Nauru
Northern Mariana Islands                    Northern Mariana Islands
Palau                                                          Palau
                                 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
                                                   Name value
American Samoa                           American Samoa  1.00
Cook Islands                               Cook Islands  1.90
French Polynesia                       French Polynesia 11.06
Niue                                               Niue 22.07
Samoa                                             Samoa 27.30
Tonga                                             Tonga  1.98
Tuvalu                                           Tuvalu 11.76
Wallis and Futuna Islands Wallis and Futuna Islands  0.00
```

2.4.2 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"` 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"
```

2.5 Using `rrapply` on data.frames

In the previous section, the `dfAsList` argument is used in order to avoid recursing into the individual columns of a data.frame object. However, it can also be useful to exploit exactly this property of base `rapply`. A convenient way to apply a function to columns of a data.frame of a certain class is through the use of the `classes` argument in base `rapply`.

For instance, suppose we wish 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` symbol:

```
> iris_standard_sepal <- rrapply(
    iris,
    condition = function(x) 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
```

2.5.1 Computational effort

As `rrapply` is written in R's internal C API, it is computationally much more efficient than its equivalent implementation based on recursion through a nested list in interpreted R. The computational efficiency of `rrapply` is illustrated in the figures below, which show several benchmark timings of `rrapply` against that of base `rapply` and common `data.table` and `dplyr`-alternatives in the context of data.frame manipulation. In particular, we time the application a dummy function `f` (unit multiplication) respectively to each column and to each *numeric* column of a data.frame `x` of size $(M \times N)$, where the entries of the data.frame `x` are randomly sampled from a uniform distribution on the unit interval.

The displayed timings are the median computation times (ms) of 100 evaluations of the benchmarked expressions on a single-core processor (Intel i7-8550U, 1.80 GHz, 16 GiB system memory). In the left-hand plots, the number of data.frame rows is fixed at $M = 1000$ and the number of columns increases from $N = 100$ to $N = 10\ 000$, and in the right-hand plots the number of data.frame columns is fixed at $N = 100$ and the number of rows increases from $M = 1000$ to $M = 10^6$.

The benchmarked expressions to mutate all columns of the data.frame `x` evaluated in the first figure are:

```
> ## rapply(how = "replace")
> rapply(x, f = `*`, e2 = 1, how = "replace")
> ## rrapply(how = "replace")
> rrapply(x, f = `*`, e2 = 1, how = "replace")
> ## rrapply(how = "prune")
> rrapply(x, f = `*`, e2 = 1, how = "prune")
> ## data.table::set
> for(j in 1:length(x)) data.table::set(x, j = j, value = `*`(x[[j]], 1))
> ## dplyr::mutate_all
> dplyr::mutate_all(x, .funs = `*`, e2 = 1)
```

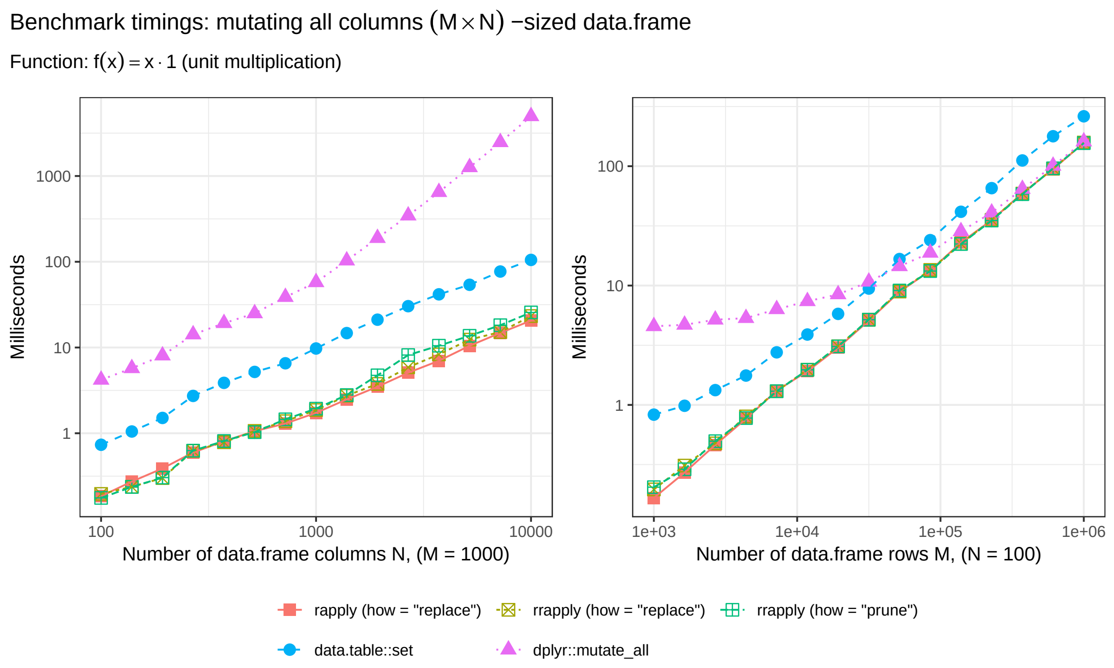
Remark. The for-loop combined with `data.table::set` was slightly faster than other alternatives using e.g. the `:=` operator. In addition, the data.frame `x` was first converted to a `data.table` object, the time to convert the object is not included in the computation times in the figures.

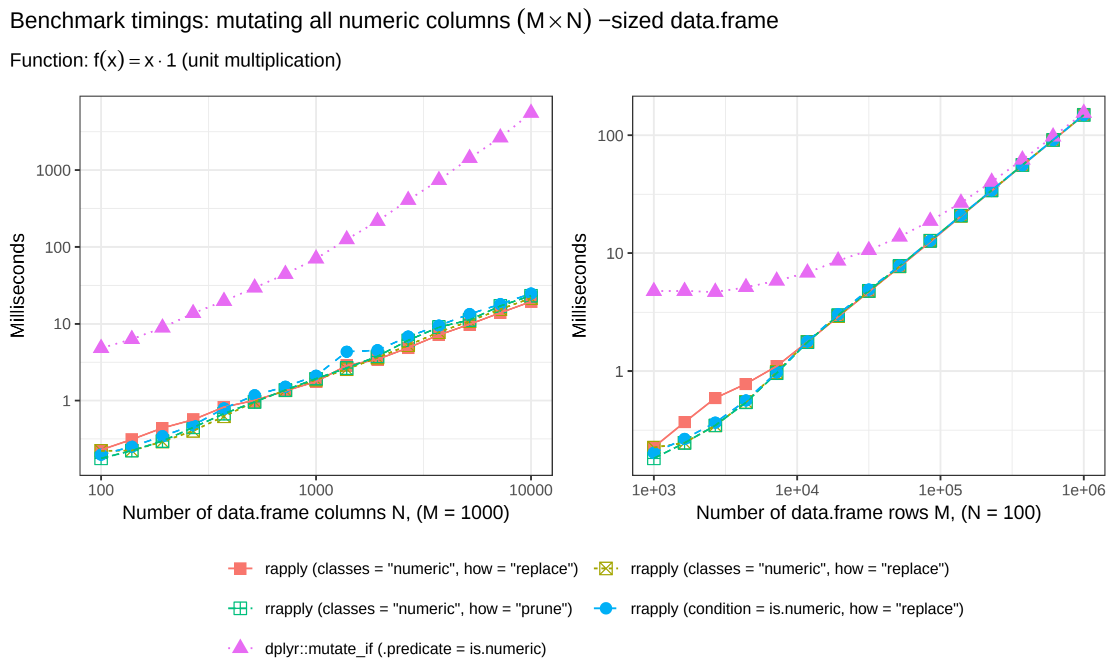
The benchmarked expressions to mutate all *numeric* columns of the data.frame `x` evaluated in the second figure are:

```
> ## rapply(classes = "numeric", how = "replace")
> rapply(x, f = `*`, e2 = 1, classes = "numeric", how = "replace")
> ## rrapply(classes = "numeric", how = "replace")
> rrapply(x, f = `*`, e2 = 1, classes = "numeric", how = "replace")
> ## rrapply(classes = "numeric", how = "prune")
> rrapply(x, f = `*`, e2 = 1, classes = "numeric", how = "prune")
> ## rrapply(condition = "numeric", how = "replace")
> rrapply(dat, condition = is.numeric, f = function(x) `*`(x, 1), how = "replace")
> ## dplyr::mutate_if(.predicate = is.numeric)
> dplyr::mutate_if(x, .predicate = is.numeric, .funs = `*`, e2 = 1)
```

Remark. Note that no `data.table`-alternative is included in the second figure as there is no direct approach to mutate a selection of data.frame columns based on a condition without relying on, for instance, an additional call to `sapply` or similar to evaluate the condition on each data.frame column.

We observe from the figures that the `rrapply` and `rapply` function calls all result in roughly similar processing times and are in general somewhat more efficient than their `data.table` and `dplyr`-alternatives. The likely reason for this is that the `rrapply` and `rapply` implementations are more basic and smaller in scope than their `data.table` and `dplyr`-alternatives, resulting in a smaller overhead than the function implementations in `data.table` and `dplyr`.


Benchmark timings: mutating all columns $(M \times N)$ −sized data.frame
Function: $f(x) = x \cdot 1$ (unit multiplication)
Milliseconds
1000
100
10
1
100
1000
10000
Number of data.frame columns N, (M = 1000)
Milliseconds
100
10
1
1e+03
1e+04
1e+05
1e+06
Number of data.frame rows M, (N = 100)
rapply (how = "replace")
rrapply (how = "replace")
rrapply (how = "prune")
data.table::set
dplyr::mutate_all


Benchmark timings: mutating all numeric columns $(M \times N)$ −sized data.frame
Function: $f(x) = x \cdot 1$ (unit multiplication)
Milliseconds
1000
100
10
1
100
1000
10000
Number of data.frame columns N, (M = 1000)
Milliseconds
100
10
1
1e+03
1e+04
1e+05
1e+06
Number of data.frame rows M, (N = 100)
rapply (classes = "numeric", how = "replace")
rrapply (classes = "numeric", how = "replace")
rrapply (classes = "numeric", how = "prune")
rrapply (condition = is.numeric, how = "replace")
dplyr::mutate_if (.predicate = is.numeric)