More details about the compare package

Paul Murrell

September 9, 2024

This document provides more information about how the **compare** package works. It should only be read after the accompanying vignette "Introduction to the compare package".

We will look in more detail at how the `compare()` function works, plus we will explore some of the lower-level functions that the `compare()` function calls.

The `"comparison"` class

The value returned by `compare()` is an object of class `"comparison"`. Importantly, this is not simply a logical value. In order to test whether the comparison was successful overall, for example as the condition in an `if` statement, you must use the `isTRUE()` function.

```
> isTRUE(compare(1:10, 1:10))

[1] TRUE

>
```

The `transforms()` function can be used to extract the vector of transformations from a `"comparison"` object.

```
> obj1 <- c("a", "a", "b", "c")
> obj1

[1] "a" "a" "b" "c"

>

> obj2 <- factor(obj1)
> obj2

[1] a a b c
Levels: a b c

>

> transforms(compare(obj1, obj2[1:3], allowAll=TRUE))

[1] "coerced from <factor> to <character>"
[2] "shortened model"

>
```

Order and persistence of transformations

The `compare()` function performs transformations in a particular order.

Any rounding of numeric values, trimming or upper-casing of strings, sorting or dropping of factor levels, reordering of dimensions for arrays, matrices, and tables, and reordering of data frame columns or list components occurs as part of the test for equality. This means that these transformations are only available if `equal=TRUE`, but also means that these transformations will be applied again after every other transformation that is attempted.

The remaining transformations occur in the following order: coerce, shorten, ignore sort order, ignore case of names, ignore names altogether, ignore attributes altogether. The justification for this order is that sorting only makes sense once the objects are of the same class and size, attributes are less fundamental than the core data structure so they should come later, and names are just a special case of general attributes, so they should come earlier.

In general, a transformation is persistent. For example, if coercion has been attempted, but the objects being compared are not the same after coercion, then sorting the objects will be attempted and *this sorting will occur on the coerced objects.* The exception to this rule is that any transformations applied during a test for equality are *not* persistent. This is because tests for equality are repeated after every other transformation.

The following manufactured example demonstrates these rules about the order and persistence of transformations. In this case, the comparison object is a different class from the model object, the comparison is in a different order from the model, *and* the comparison needs to be rounded to be equal to the model.

```
> compare(as.numeric(1:10),
+         as.character(10:1 + .1),
+         round=TRUE, coerce=TRUE, ignoreOrder=TRUE)

TRUE
  coerced from <character> to <numeric>
  sorted
  rounded

>
```

First of all, the objects are tested for equality (`equal=TRUE` by default). In this case, despite the fact that `round=TRUE`, no rounding occurs because the comparison is not numeric.

The objects are not the same, but `coerce=TRUE` so the comparison object is coerced to a numeric object *and the two objects are checked again for equality.* Furthermore, because `round=TRUE`, the coerced comparison values are rounded as well.

The objects are still not the same, but because `ingoreOrder=TRUE` the comparison object is now sorted. This sorting occurs on the *coerced* but *not rounded* comparison object because equality transformations are *not* persistent, but all other transformations *are* persistent. The coerced and sorted comparison object is then tested for equality with the model object, which again involves rounding the comparison object, and this time the objects are the same.

The overall result is success and the transformations that lead to success were coercion, followed by sorting, followed by rounding.

If the order imposed by `compare()` is not appropriate, it is possible to resort to performing the transformations individually (see the next section).

Extending the compare package

The `compare()` function is built on a set of functions that perform the individual transformations. This means that it should be relatively straightforward to produce an alternative to `compare()` by simply reordering the calls to transform functions *and* it should be relatively easy to extend the comparisons by writing new transformation functions.

Standalone comparisons

The `compare()` function is built upon a set of standalone comparison functions. Table 1 lists the standalone functions currently available.

The `compareIdentical()` function is just a wrapper for `identical()` that returns a `"comparison"` object as the result (so that it can play nicely with the other comparison functions.

The `compareEqual()` function relaxes the comparison (depending on the arguments it is given) and allows for minor differences.

All of the other comparison functions call `compareIdentical()`, and if that fails, and `equal=TRUE`, they call `compareEqual()`. These functions allow individual transformations to be attempted in isolation. A basic design feature of these functions is that they attempt to check whether they need to perform their transformation before they do it so that only necessary or potentially beneficial trasformations are attempted. For example, if two objects are of the same class, `compareCoerce()` will *not* attempt to coerce the comparison object.

A more general design feature is that the transformations have been broken into separate functions on the basis of orthogonality. Every effort is made to make the transformations independent in the sense that one transformation does not make any subsequent transformation impossible or nonsensical. As an example, a transformation that sorts objects is completely compatible with later dropping the names attributes from the objects.

All of these comparison functions are generic so that appropriate methods can be written for different classes. There are methods for the basic vector types and for arrays, matrices, tables, data frames, and lists. By having them as standalone functions, new methods can easily be developed and incorporated.

In general, comparisons involving recursive objects (data frames and lists) will transform columns or components instead of or as well as transforming the overall object. For example, `compareCoerce().data.frame` coerces the overall comparison object to a data frame *and* all columns of the comparison to the same classes as the corresponding columns of the model data frame.

These recursive comparison methods should provide the option of reordering the columns or components by name before performing transformations on the columns or components.

All comparisons other than `compareIdentical()` and `compareEqual()` should return a "partial" result as well as a full record of transformations and transformed objects. This partial result will contain transformations and transformed objects from the primary transformation for that function, but *not* any subsequent transformations due to `compareEqual()`. This allows for calling a sequence of standalone comparison functions without repeatedly recording unsuccessful transformations performed by `compareEqual()`. The utility function `same()` takes care of this issue automatically.

More about the `"comparison"` class

In addition to the overall result and the transformations attempted during a comparison, the `"comparison"` class also contains a copy of the transformed model and comparison objects.

Table 1: Standalone comparison functions upon which the compare() function is built.

Name	Description
`compareIdentical()`	Test whether two objects are identical.
`compareEqual()`	Compare whether two objects are equal. Rounds numeric values, trims leading and trailing whitespace and ignores case in strings, drops unused levels and ignores level order in factors, ignores dimension order in arrays (and matrices and tables), orders columns or components by name (ignoring case) for lists and data frames.
`compareCoerce()`	If necessary, coerce comparison object to class of model object, then compare for equality. Orders columns or components by name (ignoring case) for lists and data frames.
`compareShorten()`	If necessary, shorten the longer of the model and comparison objects so that the two objects are the same "size". For arrays, drops entire dimensions, for matrices, forces comparison to be two-dimensional, and for tables, collapses (sums) across extra dimensions. For data frames will drop columns and rows. Orders columns or components by name (ignoring case) for lists and data frames.
`compareIgnoreOrder()`	If necessary, sort both model and comparison objects, then compare for equality. Orders columns or components by name (ignoring case) for lists and data frames.
`compareIgnoreNameCase()`	If necessary, upper cases name attributes of both model and comparison, then compare for equality. For data frames, upper cases rownames as well as column names. Orders columns or components by name (ignoring case) for lists and data frames.
`compareIgnoreNames()`	If necessary, drops name attributes from both model and comparison then compare for equality. Orders columns or components by name (ignoring case) for lists and data frames.
`compareIgnoreAttrs()`	If necessary, drop all attributes from both model and comparison, then compare for equality.

These objects provide a record of what the original objects look like after a *successful* transformation (i.e., when the transformed objects are equal).

In addition to this information, a set of "partial" results are also stored. These are the transformations, plus the transformed model and comparison objects, *without* any transformations that were carried out as part of the test for equality. These objects are useful following an *unsuccessful* comparison, as the basis for further transformations.

Combining comparisons

The `compare()` function works by calling the standalone comparison functions in sequence until a successful result is achieved (or all possible transformations have been attempted). The calls are arranged in the following general format:

```
comp <- comparisonA(model, object, ...)
if (!comp$result && doComparisonB) {
    comp <- comparisonB(comp$tMpartial,
                        comp$tCpartial,
                        comp$partialTransform,
                        ...)
}
```

If the first comparison is successful, the full result is returned, so all transformations, including anything by `compareEqual()` are reported. However, if the first comparison fails, only the partial result is passed on; `compareEqual()` will be called again and will have another chance to try all of its transformations.

This basic pattern can easily be implemented "by hand" to produce a specific subset and a specific ordering of standalone comparisons, or as the basis of a variation on the `compare()` function. It is also a simple pattern to follow if new transformations need to be inserted into or added onto the current set available in `compare()`.

A worked example

This section describes the addition of comparison methods for `"list"` objects to give an idea of the design considerations that have gone into the existing code and as a template for the possible addition of methods for further classes. This should be read in conjunction with the relevant source code.

There is no need for an `identical.list()` method. The default, which calls `identical()` works already for lists and does the appropriate thing; tests whether the model and comparison objects are exactly the same.

The `compareEqual()` method differs from `compareIdentical()` because it allows the model and comparison objects to have minor differences. In the case of a list, this means that we will allow two lists to be the equal if they consist of the same number of components *and* the names of the components are the same, even though they may be in a different order and they may even differ in terms of case. Of course, the actual components themselves must also be equal, so this function calls `compareEqual()` for each pair of model-component and comparison-component. For additional flexibility, the argument `recurseFun` actually specifies which function is called on the pairs of components. This allows `compare()` to specify itself as the recursion function. The relaxation in terms of order and case of component names is not "on" by default; arguments are provided to enable these relaxations. NOTE that the uppercasing of names and reordering of

names is NOT recorded in the partial results (i.e., these transformations are NOT persistent). The result is a `"multipleComparison"`, which includes not only the overall result, but also a breakdown per component of which components were equal.

The first step in `compareCoerce.list()` is to make sure that the comparison object is a list, coercing if necessary. After that, things run very similarly to `compareEqual()`; we reorder components by name if possible and necessary, then we attempt to compare all components of the model and comparison, coercing components as necessary. The one major difference is that any transformations—coercion and reordering of components—*are* persistent this time, so become part of the partial result.

The `compareShorten()` method starts to get a bit easier. Again, we reorder components if necessary, but then all we need to do is drop any extra components and call the `same()` function to do the identical/equal check on the resulting model and comparison (which are now of the same length).

`compareIgnoreOrder()` is even easier; we just reorder the components as before, then call `same()`. Likewise `compareIgnoreNameCase`. For `compareIgnoreNames()` there's just the additional action of dropping names attributes *after* ordering by name (in case *some* names are in common, but other names conflict).

The default `compareIgnoreAttrs()` will do for lists.

Finally, for the `compare()` function, we need to add the `ignoreComponentOrder` argument so that this can be included in a general comparison, with this argument defaulting to the current setting of `allowAll`.