Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. http://www.jstatsoft.org/

Generalized and Customizable Sets in R

David Meyer
Wirtschaftsuniversität Wien

Kurt Hornik
Wirtschaftsuniversität Wien

Abstract

We present data structures and algorithms for sets and some generalizations thereof (fuzzy sets, multisets, and fuzzy multisets) available for R through the **sets** package. Fuzzy (multi-)sets are based on dynamically bound fuzzy logic families. Further extensions include user-definable iterators and matching functions.

Keywords: R, set, fuzzy logic, multiset, fuzzy set.

1. Introduction

Only few will deny the importance of sets and set theory, building the fundamentals of modern mathematics. For theory-building typically axiomatic approaches (e.g., Zermelo 1908; Fraenkel 1922) are used. However, even the primal, "naive" concept of sets representing "collections of distinct objects" (Cantor 1895) discarding order and count information seems both natural and practical. The main operation being "is-element-of", sets alone are of limited *practical* use—they most of the times serve as basic building blocks for more complex structures such as relations and generalized sets. A common way is to consider pairs (X, m) with set X ("universe") and membership function $m : X \to D$ mapping each member to its "grade". The subset of X of elements with non-zero membership is called "support". In *multisets*, elements may appear more than once, i.e., $D = \mathbb{N}_0$ (m is also called the multiplicity function). There are many applications in computer science and other disciplines (for a survey, see, e.g., Singh, Ibrahim, Yohanna, and Singh 2007). In statistics, multisets appear as frequency tables. *Fuzzy sets* have become quite popular since their introduction by Zadeh (1965). Here, the membership function maps into the unit interval. An interesting characteristic of fuzzy sets is that the actual behavior of set operations depends on the underlying fuzzy logic employed, which can be chosen according to domain-specific needs. Fuzzy sets are actively used in fields such as machine learning, engineering, medical science, and artificial intelligence (Dubois, Prade, and Yager 1996). *Fuzzy multisets* (Yager 1986) combine both approaches by allowing each element to map to more than one fuzzy membership grade, i.e., D is the power set of

multisets over the unit interval. Examples for the application of fuzzy multisets can be found in the field of information retrieval (e.g., Matthé, Caluwe, de Tré, Hallez, Verstraete, Leman, Cornelis, Moelants, and Gansemans 2006).

The use of sets and variants thereof is common in modern general purpose programming languages: Java and C++ provide corresponding abstract data types (ADTs) in their class libraries, Pascal and Python offer sets as native data type. Surprisingly enough, sets are not standard in many mathematical programming environments such as Matlab and Mathematica, and also R. Although the two latter offer set operations such as union and intersection, these are applied to linearly indexable structures (lists and vectors, respectively), *interpreting* them as sets. When it comes to R, this emulation is far from complete, and occasionally leads to inconsistent behavior. First of all, the existing infrastructure has no clear concept of how to compare elements, leading to possibly confusing results when different data types are involved in computations:

```
> s <- list(1, "1")
> union(s, s)

[[1]]
[1] 1

[[2]]
[1] "1"

> intersect(s, s)

[[1]]
[1] 1
```

The reason is that most of the existing operations rely on `match()` which automatically performs type conversions disturbing in this context. Also, quite a few other basic operations such as the Cartesian product, the power set, the subset predicate, etc., are missing, let alone more specialized operations such as the closure under union or intersection. Then, the current facilities do not make use of a class system, making extensions hard (if not impossible). Another consequence is that no distinction can be made between sequences (ordered collections of objects) and sets (unordered collections of objects), which is key for the definition of complex structures where both concepts are combined such as relations. Also, there is no support in base R for extensions such as fuzzy sets or multisets.

A few extension packages available from CRAN deal with fuzzy concepts: Package **fuzzyFDR** (Lewin 2007) calculates fuzzy decision rules for multiple testing, but does not provide any explicit data structures for fuzzy sets. The main functions in **fso** (Roberts 2007) for fuzzy set ordination compute and return, among other information, membership values represented by numeric matrices for some variables of the the input data. **fuzzyRankTests** (Geyer 2007) provides statistical tests based on fuzzy p values and fuzzy confidence intervals, the latter being returned as two separate numeric vectors for values and memberships. The **gcl** package (Vinterbo 2007) infers fuzzy rules from the input data, encapsulated in a classifying function returned by the training function. The rules are composed of triangular fuzzy sets, represented

by triples describing the triangles' corner points for which the memberships become $(0, 1, 0)$, respectively. Similarly, the **FKBL** package for fuzzy knowledge base learning (Alvarez 2007) uses sequences of triangular fuzzy sets, defined by a vector of corner points. Finally, **fuzzyOP** (Semagul, Emine, Rabiye, Senay, and Hatice 2008) provides support for fuzzy numbers: A set of n numbers is represented by a $k \times 2n$ numeric matrix, where two consecutive columns represent (at most k) supporting points and memberships, respectively, of the corresponding piecewise linear membership function. If some numbers have fewer supporting points than others, the remaining cells are filled with missing values (`NA`s).

The **sets** package (Meyer and Hornik 2008) presented here provides a flexible and customizable basic infrastructure for *finite* sets and the generalizations mentioned above, including basic operations for fuzzy logic. Apart from complementing the data structures implemented in base R, extension packages like the ones mentioned above could gain in flexibility from building on a common infrastructure, facilitating data exchange and leveraging synergies.

The remainder of the paper is structured as follows. In Section 2, we discuss the design rationale of data structures and core algorithms. Section 3 introduces the most important set operations. Section 4 starts with constructors and specific methods for generalized sets, followed by a more focused presentation of the fuzzy logic infrastructure, and of functionality for handling and visualizing membership information. Section 5 shows how generalized sets can further be customized by specifying user-definable matching functions and iterators. Section 6 presents three examples before Section 7 concludes.

2. Design issues

There are many ways of implementing sets. Choice and efficiency largely depend on the domain range (i.e., the number of possible values for each element). If the domain is relatively small, i.e. in the range of integral data types such as `byte`, `integer`, `word` etc., the probably most efficient representation is an array of bits representing the domain elements like in Pascal (Wirth 1983). Operations such as union and intersection can then straightforwardly be implemented using logical `OR` and `AND`, respectively. This approach obviously fails for intractably large domains (e.g., strings or recursive objects). Without further application knowledge, one needs to resort to generic container ADTs with efficient element access such as hash tables or search trees (for unique elements). Operations can then be implemented following the classical element-based definitions: Union by inserting all elements of the smaller set into the larger one; intersection by creating a new set with all elements of the smaller set also contained in the larger one; etc.

Clearly, set comparison must be permutation invariant. Some care is needed for nested sets. Assume, e.g., the comparison of $A = \{1, \{2, 3\}\}$ and $B = \{1, \{3, 2\}\}$ which clearly are identical. To implement set equality, a matching operator would be used to check if all elements of A are contained in B. If elements were internally stored in this order during creation, the objects representing $\{2, 3\}$ and $\{3, 2\}$ would be different. Comparing two set elements for equality would thus require to recursively compare all elements down the nested structures, which can quickly become infeasible computationally. We avoid this by using a canonical ordering during set creation, guaranteeing that identical sets have identical physical representation as well. We chose to sort elements using the natural order for numeric values, the Unicode character representation for strings, and the serialization byte sequence (as strings) for other

objects. Eventually, the ordered elements are stored in a list.

For the **sets** package, further limitations are imposed by the extensions presented in Sections 4 and 5: Generalized sets require, for each element, the membership information, and we also support user-defined, high-level matching functions for comparing elements. Since operations defined for generalized sets basically operate on the memberships, it seems appropriate to store these as (generic) vectors in the same order than the corresponding elements. Thus, memberships of separate sets can simply be combined element-wise.

Many operations (e.g., testing for equality, subsetting, intersection, etc.) are based on matching elements of the sets involved. This is implemented by inserting the elements of the larger one into a hash table (we use hashed environments), and to look up the elements of the smaller set in this table (Knuth 1973, p. 391). As hash key, we use the elements' character representation. Since different objects might map to the same hash key, we actually store the *indexes* of the list elements, and match the actual objects using a simple linear search. (Note that since the element list is sorted, elements with same representation are grouped, so the search will typically be fast.)

Objects for sets, generalized sets and customizable sets have S3 classes set, gset and cset, respectively, with set inheriting from gset in turn inheriting from cset. Accordingly, all operations have `set_`, `gset_` or `cset_` prefixes, respectively, to give the user the choice of up- or downcasts when objects of different class levels are involved in one computation. For example, consider the union of the set $\{1\}$ and the fuzzy set $\{2/0.5\}$: If the result should be a generalized (fuzzy) set, `gset_union()` should be used. To make the result a set (stripping membership information), one employs `set_union()` instead.

3. Sets

The basic constructor for creating sets is the `set()` function accepting any number of R objects as arguments.

```
> s <- set(1L, 2L, 3L)
> print(s)

{1L, 2L, 3L}
```

For elements that are not sets or atomic vectors of length 1, the print method for sets will use labels indicating the class (and length for vectors):

```
> set("test", c, set("a", 2.5), list(1, 2))

{"test", <<function>>, {"a", 2.5}, <<list(2)>>}
```

Mainly for cosmetic reasons, there is also a tuple class that can be used for vectors:

```
> set(1, pair(1, 2), tuple(1, 2, 3))

{1, (1, 2), (1, 2, 3)}
```

In addition, there is a generic `as.set()` function coercing suitable objects to sets.

```
> s2 <- as.set(2:4)
> print(s2)

{2L, 3L, 4L}
```

There are some basic predicate functions (and corresponding operators) defined for the (in)equality (`!=`, `==`), (proper) subset (`<`, `<=`), (proper) superset (`>`, `>=`), and element-of (`%e%`) operations:

```
> set_is_empty(set())

[1] TRUE

> set_is_subset(set(1), set(1, 2))

[1] TRUE

> set(1) <= set(1, 2)

[1] TRUE
```

Note that all predicate functions are vectorized:

```
> set_contains_element(set(1, 2, 3), 1)

[1] TRUE

> 1:4 %e% set(1L, 2L, 3L)

[1]  TRUE  TRUE  TRUE FALSE
```

The sequence `1:4` as *one* element would be looked up by using `list(1:4)` on the left-hand side. Other than these functions and operators, one can use `length()` for the cardinality:

```
> length(s)

[1] 3
```

`c()` and `|` for the union, `&` for the intersection, `%D%` for the symmetric difference:

```
> s | set("a")

{"a", 1L, 2L, 3L}
```

```
> s & s2

{2L, 3L}

> s %D% s2

{1L, 4L}
```

`*` and `^n` for the (n-fold) Cartesian product (yielding a set of n-tuples):

```
> s * s2

{(1L, 2L), (1L, 3L), (1L, 4L), (2L, 2L), (2L, 3L), (2L, 4L), (3L, 2L),
 (3L, 3L), (3L, 4L)}

> s^2L

{(1L, 1L), (1L, 2L), (1L, 3L), (2L, 1L), (2L, 2L), (2L, 3L), (3L, 1L),
 (3L, 2L), (3L, 3L)}
```

and `2^` for the power set:

```
> 2^s

{{}, {1L}, {2L}, {3L}, {1L, 2L}, {1L, 3L}, {2L, 3L}, {1L, 2L, 3L}}
```

`set_union()`, `set_intersection()`, and `set_symdiff()` accept more than two arguments.[1] `set_combn()` returns the set of all subsets of specified length:

```
> set_combn(s, 2L)

{{1L, 2L}, {1L, 3L}, {2L, 3L}}
```

`closure()` and `reduction()` compute the closure and reduction under union or intersection for a set *family* (i.e., a set of sets):

```
> cl <- closure(set(set(1), set(2), set(3)), "union")
> print(cl)

{{1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

> reduction(cl, "union")

{{1}, {2}, {3}}
```

[1] The n-ary symmetric difference of a collection of sets consists of all elements contained in an odd number of the sets in the collection.

The `Summary()` group methods will also work if defined for the elements:

```
> sum(s)

[1] 6

> range(s)

[1] 1 3
```

Because set elements are unordered, it is not allowed to use positional subscripting. However, sets can be subset and elements be replaced by using the elements as index themselves:

```
> s2 <- set(1, 2, c, list(1, 2))
> print(s2)

{<<function>>, 1, 2, <<list(2)>>}

> s2[[c]] <- "foo"
> s2[[list(1, 2)]] <- "bar"
> print(s2)

{"bar", "foo", 1, 2}

> s2[list("foo", 1)]

{"foo", 1}
```

Further, iterations over *all* elements can be carried out using `for()` and `lapply()`/`sapply()`:

```
> sapply(s, sqrt)

[1] 1.000000 1.414214 1.732051

> for (i in s) print(i)

[1] 1
[1] 2
[1] 3
```

Note that `for()` only works because the underlying C code ignores the class information, and directly processes the low-level list representation instead. This will be replaced by a more intelligent "foreach" mechanism as soon as it exists in base R. `sapply()` and `lapply()` call the generic `as.list()` function before iterating over the elements. Since a corresponding method exists for sets objects, this is "safer" than using `for()`.

Using `set_outer()`, it is possible to apply a function on all factorial combinations of the elements of two sets. If only one set is specified, the function is applied on all pairs of this set.

```
> set_outer(set(1, 2), set(1, 2, 3), "/")

  1   2         3
1 1 0.5 0.3333333
2 2 1.0 0.6666667
```

4. Generalized sets

There are several extensions of sets such as *fuzzy sets* and *multisets*. Both can be be seen as special cases of *fuzzy multisets*. We present how they are constructed, and demonstrate the effect of choosing different fuzzy logic families.

4.1. Constructors and specific methods

Generalized sets are created using the `gset()` function, expecting support and membership information. This can be done in four ways:

1. Specify the support only (this yields a "classical" set).
2. Specify support and memberships.
3. Specify support and membership function.
4. Specify a set of elements along with their individual membership grades, using the element function (`e()`).

Note that by default, for efficiency reasons, `gset()` will not store elements with zero memberships grades. The specification of an universe is thus only mandatory with membership functions. A default universe can be set using `sets_options()`.

Without membership information, `gset()` creates a set (the support is converted to a set internally):

```
> X <- c("A", "B", "C")
> gset(support = X)

{"A", "B", "C"}
```

A multiset requires an integer membership vector:

```
> multi <- 1:3
> gset(support = X, memberships = multi)

{"A" [1], "B" [2], "C" [3]}
```

For fuzzy sets, the memberships need to be out of the unit interval:

```
> ms <- c(0.1, 0.3, 1)
> gset(support = X, memberships = ms)

{"A" [0.1], "B" [0.3], "C" [1]}
```

Alternatively to separate support / membership specification, each elements can be paired with its membership value using `e()`:

```
> gset(elements = list(e("A", 0.1), e("B", 0.2), e("C", 0.3)))

{"A" [0.1], "B" [0.2], "C" [0.3]}
```

Fuzzy sets can, additionally, be created using a membership function, applied to a specified (or the default) universe:

```
> f <- function(x) switch(x, A = 0.1, B = 0.2, C = 1, 0)
> gset(universe = X, charfun = f)

{"A" [0.1], "B" [0.2], "C" [1]}
```

For fuzzy multisets, the membership argument expects a list of membership grades, either specified as vectors, or as multisets:

```
> ms2 <- list(c(0.1, 0.3, 0.4), c(1, 1), gset(support = ms, memberships = multi))
> gset(support = X, memberships = ms2)

{"A" [{0.1, 0.3, 0.4}], "B" [{1 [2]}], "C" [{0.1 [1], 0.3 [2], 1 [3]}]}
```

`gset_cardinality()` returns the (relative) cardinality of a generalized set, computed as the sum (mean) of all memberships. `gset_support()`, `gset_memberships()`, `gset_height()` and `gset_core()` can be used to retrieve support, memberships, height (maximum membership degree), and the core (elements with membership 1), respectively, of a generalized set. `gset_charfun()` returns a (point-wise defined) characteristic function for a given gset. Note that in general, this will be different from the characteristic function possibly used for the creation.

As for sets, the usual operations such as union and intersection are available:

```
> X <- gset(c("A", "B", "C"), 4:6)
> Y <- gset(c("B", "C", "D"), 1:3)
> gset_union(X, Y)

{"A" [4], "B" [5], "C" [6], "D" [3]}

> gset_intersection(X, Y)

{"B" [1], "C" [2]}
```

Additionally, the sum and the difference of sets are defined, which add and subtract multiplicities (or memberships for fuzzy sets):

```
> gset_sum(X, Y)

{"A" [4], "B" [6], "C" [8], "D" [3]}

> gset_difference(X, Y)

{"A" [4], "B" [4], "C" [4]}
```

For fuzzy (multi-)sets, not only the relative, but also the absolute complement is defined:

```
> gset_complement(gset(1, 0.3))

{1 [0.7]}

> !gset(1, 0.3)

{1 [0.7]}
```

(Note the use of the `!` operator).

`gset_mean()` creates a new set by averaging corresponding memberships using the arithmetic, geometric or harmonic mean. Note that missing elements have 0 membership degree:

```
> x <- gset(1:3, 1:3/3)
> y <- gset(1:2, 1:2/2)
> gset_mean(x, y)

{1L [0.4166667], 2L [0.8333333], 3L [0.5]}

> gset_mean(x, y, "harmonic")

{1L [0.4], 2L [0.8]}

> gset_mean(set(1), set(1, 2))

{1 [1], 2 [0.5]}
```

The membership vector of a generalized set can be transformed via `gset_transform_memberships()`, applying any *vectorized* function the memberships:

```
> x <- gset(1:10, 1:10/10)
> gset_transform_memberships(x, pmax, 0.5)

{1L [0.5], 2L [0.5], 3L [0.5], 4L [0.5], 5L [0.5], 6L [0.6], 7L [0.7],
 8L [0.8], 9L [0.9], 10L [1]}
```

Note the effect of applying transformations to (multi)sets:

```
> x <- gset(1, 2)
> gset_transform_memberships(x, '*', 0.5)

{1 [{0.5 [2]}]}

> rep(x, 0.5)

{1}
```

In addition, three convenience functions exist: `gset_concentrate()` and `gset_dilate()` apply the square and the square root function, and `gset_normalize()` normalizes the memberships to a specified maximum:

```
> gset_dilate(y)

{1L [0.7071068], 2L [1]}

> gset_concentrate(y)

{1L [0.25], 2L [1]}

> gset_normalize(y, 0.5)

{1L [0.25], 2L [0.5]}
```

4.2. Fuzzy logic and fuzzy sets

For fuzzy (multi-)sets, the user can choose the logic underlying the operations using the `fuzzy_logic()` function. Fuzzy logics are represented as named lists with four components `N`, `T`, `S`, and `I` containing the corresponding functions for negation, conjunction ("t-norm"), disjunction ("t-conorm"), and (residual) implication (Klement, Mesiar, and Pap 2000). The fuzzy logic is selected by calling `fuzzy_logic()` with a character string specifying the fuzzy logic "family", and optional parameters. The exported functions `.N.()`, `.T.()`, `.S.()`, and `.I.()` reflect the currently selected bindings. Available families include: `"Zadeh"` (default), `"drastic"`, `"product"`, `"Lukasiewicz"`, `"Fodor"`, `"Frank"`, `"Hamacher"`, `"Schweizer-Sklar"`, `"Yager"`, `"Dombi"`, `"Aczel-Alsina"`, `"Sugeno-Weber"`, `"Dubois-Prade"`, and `"Yu"` (see Appendix A). A call to `fuzzy_logic()` without arguments returns the current logic.

```
> x <- 1:10/10
> y <- rev(x)
> .S.(x, y)

 [1] 1.0 0.9 0.8 0.7 0.6 0.6 0.7 0.8 0.9 1.0
```

```
> fuzzy_logic("Fodor")
> .S.(x, y)

 [1] 1 1 1 1 1 1 1 1 1 1
```

Fuzzy set operations automatically use the active fuzzy logic setting:

```
> X <- gset(c("A", "B", "C"), c(0.3, 0.5, 0.8))
> print(X)

{"A" [0.3], "B" [0.5], "C" [0.8]}

> Y <- gset(c("B", "C", "D"), c(0.1, 0.3, 0.9))
> print(Y)

{"B" [0.1], "C" [0.3], "D" [0.9]}
```

First, we try the Zadeh logic (default):

```
> fuzzy_logic("Zadeh")
> gset_intersection(X, Y)

{"B" [0.1], "C" [0.3]}

> gset_union(X, Y)

{"A" [0.3], "B" [0.5], "C" [0.8], "D" [0.9]}

> gset_complement(X, Y)

{"B" [0.1], "C" [0.2], "D" [0.9]}
```

The results are different by switching to the Fodor logic:

```
> fuzzy_logic("Fodor")
> gset_intersection(X, Y)

{"C" [0.3]}

> gset_union(X, Y)

{"A" [0.3], "B" [0.5], "C" [1], "D" [0.9]}

> gset_complement(X, Y)

{"D" [0.9]}
```

The `cut()` method for generalized sets "filters" all elements with membership not less then a specified level—the result, thus, is a crisp (multi)set:

```
> cut(X, 0.5)

{"B", "C"}
```

4.3. Characteristic functions and their visualization

The **sets** package provides several generators of characteristic functions to be used as templates for the creation of fuzzy sets, including the following shapes: gaussian curve (`fuzzy_normal()`), double gaussian curve (`fuzzy_two_normals()`), bell curve (`fuzzy_bell()`), sigmoid curve (`fuzzy_sigmoid()`), trapezoid (`fuzzy_trapezoid()`), and triangle (`fuzzy_triangular()`, `fuzzy_cone()`). For example, a fuzzy normal function and a corresponding fuzzy set are created using:

```
> N <- fuzzy_normal(mean = 0, sd = 1)
> N(-3:3)

[1] 0.01110900 0.13533528 0.60653066 1.00000000 0.60653066 0.13533528 0.01110900

> gset(charfun = N, universe = -3:3)

{-3L [0.01110900], -2L [0.1353353], -1L [0.6065307], 0L [1], 1L
 [0.6065307], 2L [0.1353353], 3L [0.01110900]}
```

For convenience, we also provide wrappers that directly generate corresponding sets, given a specified (default) universe:

```
> fuzzy_normal_gset(universe = -3:3)

{-3L [0.01110900], -2L [0.1353353], -1L [0.6065307], 0L [1], 1L
 [0.6065307], 2L [0.1353353], 3L [0.01110900]}
```

It is also possible to create function generators for characteristic functions from other functions (such as distribution functions):

```
> fuzzy_poisson <- charfun_generator(dpois)
> gset(charfun = fuzzy_poisson(10), universe = seq(0, 20, 2))

{0 [0.00036288], 2 [0.018144], 4 [0.1512], 6 [0.504], 8 [0.9], 10 [1],
 12 [0.7575758], 14 [0.4162504], 16 [0.1734377], 18 [0.05667898], 20
 [0.01491552]}
```

`fuzzy_tuple()` generates a sequence (tuple) of sets based on any of the generating functions (except `fuzzy_trapezoid()` and `fuzzy_triangular()`). The chosen generating function is called with different values (chosen along the universe) passed to the first argument, thus varying the position or the resulting graph:

```
> fuzzy_tuple(fuzzy_normal, 5)

(<<gset(201)>>, <<gset(201)>>, <<gset(201)>>, <<gset(201)>>,
 <<gset(201)>>)
```

(`<<gset(201)>>` denotes an object of class `gset` with 201 elements—the size of the default universe). The **sets** package provides support for visualizing the membership functions of generalized sets, and in particular fuzzy sets. For (fuzzy) multisets, the plot method produces a (grouped) barplot for the membership vector (see Figure 1, top left):

```
> X <- gset(c("A", "B"), list(1:2/2, 0.5))
> plot(X)
```

Characteristic function generators can directly be plotted using a default universe (see Figure 1, top right):

```
> plot(fuzzy_bell)
```

There is a plot method for tuples for visualizing a sequence of sets (see Figure 1, bottom left):

```
> plot(fuzzy_tuple(fuzzy_cone, 10), col = gray.colors(10))
```

Plots of several sets can be superposed using the line method (see Figure 1, bottom right):

```
> x <- fuzzy_normal_gset()
> y <- fuzzy_trapezoid_gset(corners = c(5, 10, 15, 17), height = c(0.7,
+     1))
> plot(tuple(x, y), lty = 3)
> lines(x | y, col = 2)
> lines(gset_mean(x, y), col = 3, lty = 2)
```

5. User-definable extensions

We added *customizable sets* extending generalized sets in two ways: First, users can control the way elements are matched, i.e., define equivalence classes of elements. Second, arbitrary iteration orders can be specified.

5.1. Matching functions

By default, sets and generalized sets use `identical()` to match elements which is maximally restrictive. Note that this differs from the behavior of R's equality operator or `match()` which perform implicit type conversions and thus confound, e.g., `1`, `1L` and `"1"`. In the following example, note that on most computer systems, $3.3 - 2.2$ will not be identical to 1.1 due to numerical issues.

```
> x <- set("1", 1L, 1, 3.3 - 2.2, 1.1)
> print(x)
```

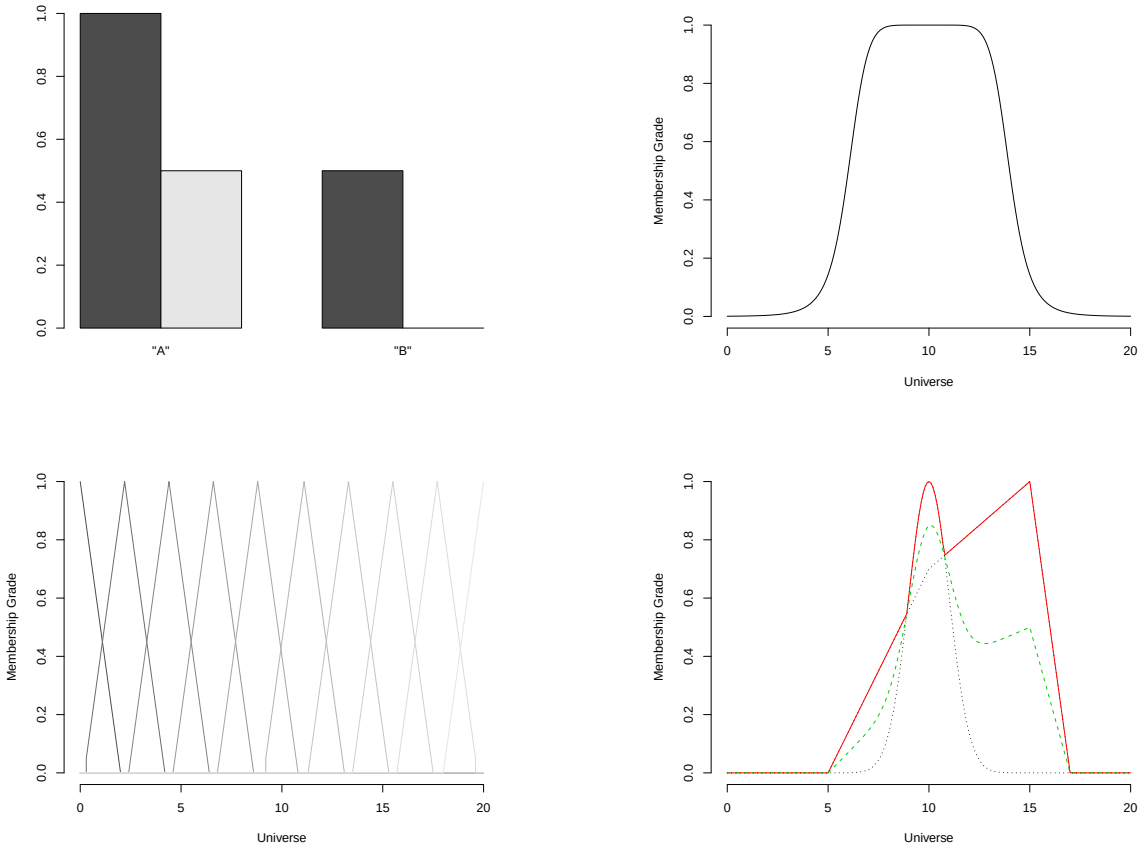

Figure 1: Membership plots for fuzzy sets. Top left: grouped barplot for a fuzzy multiset. Top right: graph of a bell curve. Bottom left: sequence of triangular functions. Bottom right: two combinations of a normal and a trapezoid function (dotted lines: basic shapes; solid (red) line: union; dashed (green) line: arithmetic mean).

```
{"1", 1L, 1, 1.1, 1.1}

> y <- set(1, 1.1, 2L, "2")
> print(y)

{"2", 2L, 1, 1.1}

> 1L %e% y

[1] FALSE

> x | y

{"1", "2", 1L, 2L, 1, 1.1, 1.1}
```

Customizable sets can be created using the `cset()` constructor, specifying the generalized set and some matching function.

```
> X <- cset(x, matchfun = match)
> print(X)

{"1", 1.1}

> Y <- cset(y, matchfun = match)
> print(Y)

{"2", 1, 1.1}

> 1L %e% Y

[1] TRUE

> X | Y

{"1", "2", 1.1}
```

Matching functions take two vector arguments, say, `x` and `table`, with `table` being a vector where the elements of `x` are looked up. The function should be vectorized in the `x`, i.e. return the first matching position for each element of `x`. In order to make use of non-vectorized predicates such as `all.equal()`, the **sets** package provides `matchfun()` to generate one:

```
> FUN <- matchfun(function(x, y) isTRUE(all.equal(x, y)))
> X <- cset(x, matchfun = FUN)
> print(X)

{"1", 1L, 1.1}

> Y <- cset(y, matchfun = FUN)
> print(Y)

{"2", 2L, 1, 1.1}

> 1L %e% Y

[1] TRUE

> X | Y

{"1", "2", 1L, 2L, 1.1}
```

`sets_options()` can be used to conveniently switch the default match and/or order function if a number of cset objects need to be created:

```
> sets_options("matchfun", match)
> cset(x)

{"1", 1.1}

> cset(y)

{"2", 1, 1.1}

> cset(1:3) <= cset(c("1", "2", "3"))

[1] TRUE
```

5.2. Iterators

In addition to specifying matching functions, it is possible to change the order in which iterators such as `as.list()` (but not `for()`—see end of Section 3) process the elements. Note that the behavior of `as.list()` influences the labeling and print methods for customizable sets. Sets and generalized sets use the canonical internal ordering for iterations. With customizable sets, a "natural" ordering of elements can be kept by specifying either a permutation vector or an order function:

```
> cset(letters[1:5], orderfun = 5:1)

{"e", "d", "c", "b", "a"}

> FUN <- function(x) order(as.character(x), decreasing = TRUE)
> Z <- cset(letters[1:5], orderfun = FUN)
> print(Z)

{"e", "d", "c", "b", "a"}

> as.character(Z)

[1] "e" "d" "c" "b" "a"
```

Note that converters for ordered factors keep the order:

```
> o <- ordered(c("a", "b", "a"), levels = c("b", "a"))
> as.set(o)

{a, b}

> as.gset(o)
```

```
{a [2], b [1]}
```

```
> as.cset(o)
```

```
{b [1], a [2]}
```

Converters for other data types will use the order information only if elements are unique:

```
> as.cset(c("A", "quick", "brown", "fox"))
```

```
{"A", "quick", "brown", "fox"}
```

```
> as.cset(c("A", "quick", "brown", "fox", "quick"))
```

```
{"A" [1], "brown" [1], "fox" [1], "quick" [2]}
```

6. Examples

In the following, we present two examples for the use of multisets and generalized sets.

6.1. Multisets

Multisets are frequent in statistics since they can be seen as frequency tables of some objects. Using the **sets** package, a "generalized" table can easily be constructed from a list of R objects using the `as.gset()` coercion function. Assume, e.g., that one samples a number of fourfold tables given the margins using `r2dtable()`:

```
> set.seed(4711)
> l <- r2dtable(1000, r = 1:2, c = 2:1)
```

Since the sum of the first row (and second column) are constrained to 1, the top left cell entry can only be 0 or 1. Also, given the marginals, there is only one degree of freedom in fourfold tables, so the value of this first cell determines the others, and thus only two possible tables exist:

```
> l[1:2]
```

```
[[1]]
     [,1] [,2]
[1,]    0    1
[2,]    2    0

[[2]]
     [,1] [,2]
[1,]    1    0
[2,]    1    1
```

To count them, we can simply use `as.gset()` that will construct a multiset from the list:

```
> s <- as.gset(l)
> print(s)

{<<2x2 matrix>> [330], <<2x2 matrix>> [670]}
```

Replace the matrices by the first cells' values:

```
> for (i in s) s[[i]] <- i[1]
> print(s)

{0L [330], 1L [670]}
```

The estimated probabilities of having 0 or 1 in the first cell can thus be obtained by:

```
> gset_memberships(s)/1000

[1] 0.33 0.67
```

The probability for 0 clearly corresponds to the p value of the corresponding Fisher test:

```
> fisher.test(l[[1]])$p.value

[1] 0.3333333
```

6.2. Fuzzy multisets

Companies typically evaluate the effectiveness of their promotional activities through surveys, where respondents state whether and to which extent they associate some attributes to the promoted products. Suppose, e.g, that a commercial for a new car conveys the idea of "sportiness" and "coolness"—the surveys will help to assess if these "messages" have really been perceived by customers. It is also common to include competing products. For each of product, customers will need to pick some attributes from a list and specify some rating. The resulting data can be represented by fuzzy multisets: one for each product, the support being the set of attributes perceived, and the memberships representing the respondents' ratings for each of the attributes. In the following, we will use some simulated data (see Appendix B for the code):

```
> set.seed(4712)
> cars <- create_products(N = 10)
> cars[1:2]

[[1]]
{"comfortable" [{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}],
 "cool" [{0.1, 0.2, 0.3, 0.4, 0.7, 1}], "fast" [{0.1, 0.3, 0.4, 0.6,
```

```
 0.8, 0.9}], "spacious" [{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9,
 1}], "sporty" [{0.2, 0.3, 0.9}], "stylish" [{0.1, 0.2, 0.3, 0.4, 0.6,
 0.8, 0.9}]}

[[2]]
{"economic" [{0.3, 0.6, 0.8}]}
```

The **sets** package implements the Jaccard similarity, defined as $|X \cap Y|/|X \cup Y|$ given two generalized sets X and Y, $|\cdot|$ denoting the cardinality for generalized sets. Using the **proxy** package (Meyer and Buchta 2008), one can calculate a similarity matrix between all products for user-defined similarity measures:

```
> library("proxy")
> sim <- simil(cars, method = gset_similarity)
```

After transformation into a dissimilarity object:

```
> d <- as.dist(sim)
```

this can be used for, e.g., hierarchical clustering to find "close" products (see the dendrogram in Figure 2):

```
> plot(hclust(d))
```

Cars 1 and 6 as well as 3 and 10 seem to "map" to similar attributes:

```
> cars[c(3, 10)]
```

```
[[1]]
{"comfortable" [{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1}], "fast" [{0.1,
 0.4, 0.7, 0.9, 1}], "spacious" [{0.1, 0.2, 0.3, 0.4, 0.5, 0.8, 1}],
 "sporty" [{0.2, 0.6, 0.7, 0.8, 0.9}]}

[[2]]
{"comfortable" [{0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9}], "economic" [{0.4,
 1}], "spacious" [{0.1, 0.2, 0.7, 0.8, 0.9}], "sporty" [{0.1, 0.2, 0.3,
 0.4, 0.5, 0.6, 0.7, 0.8, 1}], "stylish" [{0.2}]}
```

Further, if the customer data base includes consumer preferences, these can be matched to the existing products for specific recommendations in the context of one-to-one marketing campaigns. We use fuzzy sets to represent some customers' preferences:

```
> customer1 <- gset(c("sporty", "trendy"), c(0.8, 0.6))
> customer2 <- gset(c("comfortable", "spacious"), c(0.5, 0.9))
> customer_DB <- list(customer1, customer2)
```

A simple 1-nearest-neighbor approach can be used to identify the best-matching product(s): again using the **proxy** package, we compute the cross-similarities between customers and products and seek the indexes with highest similarities:

```
> max.col(simil(customer_DB, cars, gset_similarity))

[1]  5 10
```

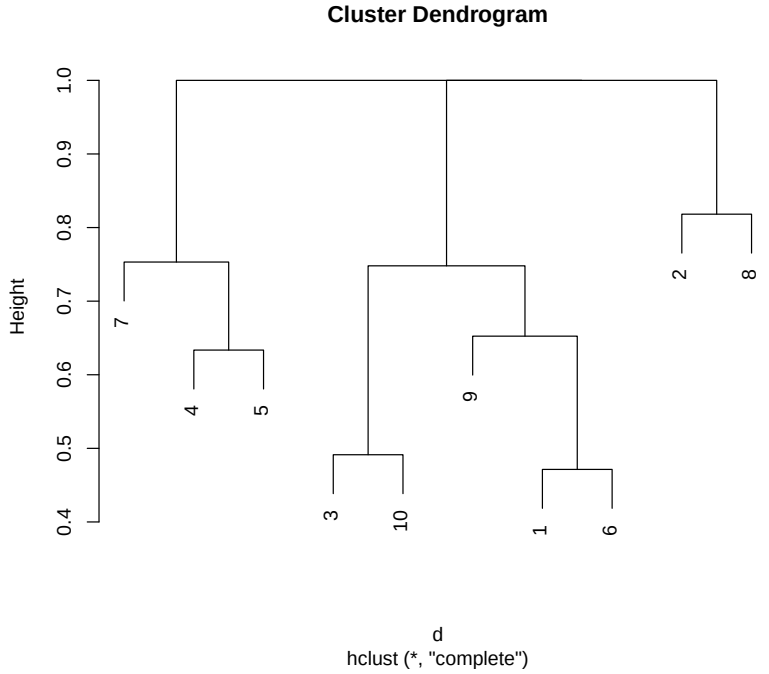

Figure 2: Dendrogram for cars data.

7. Conclusion

In this paper, we described the **sets** package for R, providing infrastructure for sets and generalizations thereof such as fuzzy sets, multisets and fuzzy multisets. The fuzzy variants make use of a dynamic fuzzy logic infrastructure offering several fuzzy logic families. Generalized sets are further extended to allow for user-defined iterators and matching functions. Current work focuses on data structures and algorithms for relations, an important application of sets.

A. Available fuzzy logic families

Let us refer to $N(x) = 1 - x$ as the *standard* negation, and, for a t-norm T, let $S(x, y) = 1 - T(1 - x, 1 - y)$ be the *dual* (or complementary) t-conorm. Available specifications and corresponding families are as follows, with the standard negation used unless stated otherwise.

`"Zadeh"` Zadeh's logic with $T = \min$ and $S = \max$. Note that the minimum t-norm, also known as the Gödel t-norm, is the pointwise largest t-norm, and that the maximum t-conorm is the smallest t-conorm.

`"drastic"` the drastic logic with t-norm $T(x, y) = y$ if $x = 1$, x if $y = 1$, and 0 otherwise, and complementary t-conorm $S(x, y) = y$ if $x = 0$, x if $y = 0$, and 1 otherwise. Note that the drastic t-norm and t-conorm are the smallest t-norm and largest t-conorm, respectively.

`"product"` the family with the product t-norm $T(x, y) = xy$ and dual t-conorm $S(x, y) = x + y - xy$.

`"Lukasiewicz"` the Łukasiewicz logic with t-norm $T(x, y) = \max(0, x + y - 1)$ and dual t-conorm $S(x, y) = \min(x + y, 1)$.

`"Fodor"` the family with Fodor's *nilpotent minimum* t-norm given by $T(x, y) = \min(x, y)$ if $x + y > 1$, and 0 otherwise, and the dual t-conorm given by $S(x, y) = \max(x, y)$ if $x + y < 1$, and 1 otherwise.

`"Frank"` the family of Frank t-norms T_p, $p \geq 0$, which gives the Zadeh, product and Łukasiewicz t-norms for $p = 0$, 1, and ∞, respectively, and otherwise is given by $T(x, y) = \log_p(1 + (p^x - 1)(p^y - 1)/(p - 1))$.

`"Hamacher"` the three-parameter family of Hamacher, with negation $N_\gamma(x) = (1-x)/(1+\gamma x)$, t-norm $T_\alpha(x, y) = xy/(\alpha + (1 - \alpha)(x + y - xy))$, and t-conorm $S_\beta(x, y) = (x + y + (\beta - 1)xy)/(1 + \beta xy)$, where $\alpha \geq 0$ and $\beta, \gamma \geq -1$. This gives a deMorgan triple (for which $N(S(x, y)) = T(N(x), N(y))$ iff $\alpha = (1 + \beta)/(1 + \gamma)$.

The following parametric families are obtained by combining the corresponding families of t-norms with the standard negation and complementary t-conorm.

`"Schweizer-Sklar"` the Schweizer-Sklar family T_p, $-\infty \leq p \leq \infty$, which gives the Zadeh (minimum), product and drastic t-norms for $p = -\infty$, 0, and ∞, respectively, and otherwise is given by $T_p(x, y) = \max(0, (x^p + y^p - 1)^{1/p})$.

`"Yager"` the Yager family T_p, $p \geq 0$, which gives the drastic and minimum t-norms for $p = 0$ and ∞, respectively, and otherwise is given by $T_p(x, y) = \max(0, 1 - ((1 - x)^p + (1 - y)^p)^{1/p})$.

`"Dombi"` the Dombi family T_p, $p \geq 0$, which gives the drastic and minimum t-norms for $p = 0$ and ∞, respectively, and otherwise is given by $T_p(x, y) = 0$ if $x = 0$ or $y = 0$, and $T_p(x, y) = 1/(1 + ((1/x - 1)^p + (1/y - 1)^p)^{1/p})$ if both $x > 0$ and $y > 0$.

`"Aczel-Alsina"` the family of t-norms T_p, $p \geq 0$, introduced by Aczél and Alsina, which gives the drastic and minimum t-norms for $p = 0$ and ∞, respectively, and otherwise is given by $T_p(x, y) = \exp(-(|\log(x)|^p + |\log(y)|^p)^{1/p})$.

`"Sugeno-Weber"` the family of t-norms T_p, $-1 \leq p \leq \infty$, introduced by Weber with dual t-conorms introduced by Sugeno, which gives the drastic and product t-norms for $p = -1$ and ∞, respectively, and otherwise is given by $T_p(x, y) = \max(0, (x+y-1+pxy)/(1+p))$.

`"Dubois-Prade"` the family of t-norms T_p, $0 \leq p \leq 1$, introduced by Dubois and Prade, which gives the minimum and product t-norms for $p = 0$ and 1, respectively, and otherwise is given by $T_p(x, y) = xy/\max(x, y, p)$.

`"Yu"` the family of t-norms T_p, $p \geq -1$, introduced by Yu, which gives the product and drastic t-norms for $p = -1$ and ∞, respectively, and otherwise is given by $T(x, y) = \max(0, (1 + p)(x + y - 1) - pxy)$.

B. Code for generating the `cars` data

```
create_products <-
function(attributes = c("comfortable", "sporty", "fast", "spacious",
         "cool", "economic", "trendy", "stylish"),
         values = 1:10 / 10,
         N = 10)
{
    sample_FUN <- function(n, values)
        lapply(sample(length(values), n, replace = TRUE),
               function(i) sample(values, i))

    attr_list <- sample_FUN(N, attributes)
    value_list <- lapply(sapply(attr_list, length), sample_FUN, values)

    Map(gset, attr_list, value_list)
}
```

References

Alvarez AG (2007). *FKBL: fuzzy knowledge base learning.* R package version 0.50-4.

Cantor G (1895). "Beiträge zur Begründung der transfiniten Mengenlehre." In "Mathematische Annalen," volume 46, pp. 481–512. Springer.

Dubois D, Prade H, Yager RY (eds.) (1996). *Fuzzy Information Engineering: A Guided Tour of Applications.* Wiley.

Fraenkel AA (1922). "Über die Grundlagen der Cantor-Zermeloschen Mengenlehre." In "Mathematische Annalen," volume 86, pp. 230–237. Springer.

Geyer CJ (2007). *fuzzyRankTests: Fuzzy Rank Tests and Confidence Intervals.* R package version 0.3-2, URL http://www.stat.umn.edu/geyer/fuzz/.

Klement EP, Mesiar R, Pap E (2000). *Triangular Norms.* Springer.

Knuth DE (1973). *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading.

Lewin A (2007). *fuzzyFDR: Exact calculation of fuzzy decision rules for multiple testing.* R package version 1.0.

Matthé T, Caluwe RD, de Tré G, Hallez A, Verstraete J, Leman M, Cornelis O, Moelants D, Gansemans J (2006). "Similarity Between Multi-valued Thesaurus Attributes: Theory and Application in Multimedia Systems." In "Flexible Query Answering systems," Lecture Notes in Computer Science, pp. 331–342. Springer.

Meyer D, Buchta C (2008). **proxy***: Distance and Similarity Measures.* R package version 0.4-1.

Meyer D, Hornik K (2008). **sets**. R package version 0.5.

Roberts DW (2007). *fso: Fuzzy Set Ordination.* R package version 1.0-1, URL http://ecology.msu.montana.edu/labdsv/R/lab11/lab11.html.

Semagul A, Emine A, Rabiye M, Senay U, Hatice U (2008). *fuzzyOP: Fuzzy numbers and the main mathematical operations.* R package version 1.0.

Singh D, Ibrahim A, Yohanna T, Singh J (2007). "An overview of the applications of multisets." *Novi Sad Journal of Mathematics*, **37**(3), 73–92.

Vinterbo SA (2007). *gcl: Compute a fuzzy rules or tree classifier from data.* R package version 1.06.5, URL http://www.r-project.org,http://www.mit.edu/~sav/fuzzy/.

Wirth N (1983). *Algorithmen und Datenstrukturen.* Teubner, Stuttgart.

Yager RR (1986). "On the theory of bags." *International Journal of General Systems*, **13**, 23–37.

Zadeh LA (1965). "Fuzzy sets." *Information and Control*, **8**(3), 338–353.

Zermelo E (1908). "Untersuchungen über die Grundlagen der Mengenlehre." In "Mathematische Annalen," volume 65, pp. 261–281. Springer.

Affiliation:

David Meyer
Department of Information Systems and Operations
E-mail: David.Meyer@wu-wien.ac.at
URL: http://wi.wu-wien.ac.at/~meyer/

Kurt Hornik
Department of Statistics and Mathematics
E-mail: Kurt.Hornik@wu-wien.ac.at
URL: http://statmath.wu-wien.ac.at/~hornik/

Wirtschaftsuniversität Wien
Augasse 2–6
1090 Wien, Austria

Journal of Statistical Software *Submitted:* yyyy-mm-dd
MMMMMM YYYY, Volume VV, Issue II. *Accepted:* yyyy-mm-dd
http://www.jstatsoft.org/