(Generalized) Sets in R

David Meyer and Kurt Hornik

2007-10-09

Abstract

This document explains algorithms and basic operations of sets and some generalizations of sets (fuzzy sets, multisets, and fuzzy multisets) available in R through the **sets** package.

There is only rudimentary support in base R for *sets*. Typically, these are represented using atomic or recursive vectors (lists), and one can use operations such as `union()`, `intersect()`, `setdiff()`, `setequal()`, and `is.element()` to emulate set operations. However, there are several drawbacks: first of all, quite a few other operations such as the Cartesian product, the power set, the subset predicate, etc., are missing. 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 relations, where both concepts are combined. Also, there is no support for extensions such as fuzzy sets or multisets. Therefore, we decided to provide more formalized and extended support for sets, and, because they are needed for Cartesian products, also for tuples.

1 Tuples

The *tuple* functions in package **sets** represent basic infrastructure for handling tuples of general (R) objects. They are used, e.g., to correctly represent Cartesian products of sets, resulting in a set of tuples (see below). Although tuple objects should behave like "ordinary" vectors for the most common operations (see examples), some functions may yield unexpected results (e.g., `table()`) or simply not work (e.g., `plot()`) since tuple objects are in fact list objects internally. There are several constructors: `tuple()` for arbitrarily many objects, and `singleton()`, `pair()`, and `triple()` for tuples of lengths 1, 2 and 3, respectively. Note that tuple elements can be named.

```
> ## constructor
> tuple(1,2,3, TRUE)

(1, 2, 3, TRUE)

> triple(1,2,3)

(1, 2, 3)

> pair(Name = "David", Height = 185)

(Name = David, Height = 185)

> tuple_is_triple(triple(1,2,3))

[1] TRUE

> tuple_is_ntuple(tuple(1,2,3,4), 4)
```

```
[1] TRUE
```

```
> ## converter
> as.tuple(1:3)
```

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

```
> ## operations
> c(tuple("a","b"), 1)
```

```
(a, b, 1)
```

```
> tuple(1,2,3) * tuple(2,3,4)
```

```
(2, 6, 12)
```

```
> rep(tuple(1,2,3), 2)
```

```
(1, 2, 3, 1, 2, 3)
```

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

```
> sum(tuple(1, 2, 3))
```

```
[1] 6
```

```
> range(tuple(1, 2, 3))
```

```
[1] 1 3
```

In addition, there is a `tuple_outer()` function to apply functions to all combinations of tuple elements. Note that `tuple_outer()` will also work for regular vectors and thus can really be seen as an extension of `outer()`:

```
> tuple_outer(pair(1, 2), triple(1, 2, 3))
```

```
  1 2 3
1 1 2 3
2 2 4 6
```

```
> tuple_outer(1:5, 1:4, "^")
```

```
   1L 2L  3L  4L
1L  1  1   1   1
2L  2  4   8  16
3L  3  9  27  81
4L  4 16  64 256
5L  5 25 125 625
```

2 Sets

The basic constructor for creating sets is the `set()` function accepting an arbitrary number of R objects as arguments (which can be named). In addition, there is a generic `as.set()` for converting suitable objects to sets.

```
> ## constructor
> s <- set(1, 2, 3)
> s
```

```
{1, 2, 3}

> ## named elements
> snamed <- set(one = 1, 2, three = 3)
> snamed

{one = 1, 2, three = 3}

> ## named elements can directly be accessed
> snamed[["one"]]

[1] 1

> ## a more complex set
> set(c, "test", list(1, 2, 3))

{test, <<function>>, <<list(3)>>}

> ## set of sets
> set(set(), set(1))

{{}, {1}}

> ## conversion functions
> s2 <- as.set(2:5)
> s2

{2L, 3L, 4L, 5L}
```

There are some basic predicate functions (and corresponding operators) defined for the (in)equality, (proper) sub-(super-)set, and element-of. Note that all the `set_is_`*foo*`()` functions are vectorized:

```
> set_is_empty(set())

[1] TRUE

> set_is_equal(set(1), set(1))

[1] TRUE

> set(1) == set(1)

[1] TRUE

> set(1) != set(2)

[1] TRUE

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

[1] TRUE

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

[1] TRUE

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

[1] TRUE
```

```
> set_is_proper_subset(set(1), set(1))

[1] FALSE

> set(1) < set(1)

[1] FALSE

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

[1] TRUE

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

[1] TRUE

> 1 %e% set(1, 2, 3)

[1] TRUE

> set_contains_element(set(1, 2, 3), 1:4)

[1] FALSE FALSE FALSE FALSE

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

[1] FALSE FALSE FALSE FALSE
```

Other than these predicate functions and operators, one can use: `c()` and `|` for the union, `-` for the difference (or complement), `&` for the intersection, `%D%` for the symmetric difference, `*` and `^`n for the (n-fold) Cartesian product (yielding a set of n-tuples), and `2^` for the power set. `set_union()`, `set_intersection()`, and `set_symdiff()` accept more than two arguments.[1] The `length` method for sets gives the cardinality. `set_combn()` returns the set of all subsets of specified length. `closure()` and `reduction()` compute the closure and reduction under union, intersection, and symmetric difference. Note that (currently) the `rep()` method for sets will just return its argument since set elements are unique.

```
> length(s)

[1] 3

> length(set())

[1] 0

> ## complement, union, intersection, symmetric difference:
> s - 1

{2, 3}

> s + set("a")

{a, 1, 2, 3}

> s | set("a")

{a, 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.

```
> s & s2

{}

> s %D% s2

{1, 2, 3, 2L, 3L, 4L, 5L}

> set(1,2,3) - set(1,2)

{3}

> set_intersection(set(1,2,3), set(2,3,4), set(3,4,5))

{3}

> set_union(set(1,2,3), set(2,3,4), set(3,4,5))

{1, 2, 3, 4, 5}

> set_symdiff(set(1,2,3), set(2,3,4), set(3,4,5))

{1, 3, 5}

> ## Cartesian product
> s * s2

{(1, 2L), (1, 3L), (1, 4L), (1, 5L), (2, 2L), (2, 3L), (2, 4L), (2,
 5L), (3, 2L), (3, 3L), (3, 4L), (3, 5L)}

> s * s

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

> s ^ 2 # same as above

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

> s ^ 3

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

> ## power set
> 2 ^ s

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

> ## subsets:
> set_combn(as.set(1:3),2)

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

```
> ## closure and reduction (under union):
> cl <- closure(set(set(1), set(2), set(3)))
> reduction(cl)

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

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

```
> sum(s)

[1] 6

> range(s)

[1] 1 3
```

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 to 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

> X <- set_outer(set(1, 2), set(1, 2, 3), set)
> X[[2, 3]]

{2, 3}

> set_outer(2^set(1, 2, 3), set_is_subset)

             {}   {1}   {2}   {3} {1, 2} {1, 3} {2, 3} {1, 2, 3}
{}         TRUE  TRUE  TRUE  TRUE   TRUE   TRUE   TRUE      TRUE
{1}       FALSE  TRUE FALSE FALSE   TRUE   TRUE  FALSE      TRUE
{2}       FALSE FALSE  TRUE FALSE   TRUE  FALSE   TRUE      TRUE
{3}       FALSE FALSE FALSE  TRUE  FALSE   TRUE   TRUE      TRUE
{1, 2}    FALSE FALSE FALSE FALSE   TRUE  FALSE  FALSE      TRUE
{1, 3}    FALSE FALSE FALSE FALSE  FALSE   TRUE  FALSE      TRUE
{2, 3}    FALSE FALSE FALSE FALSE  FALSE  FALSE   TRUE      TRUE
{1, 2, 3} FALSE FALSE FALSE FALSE  FALSE  FALSE  FALSE      TRUE
```

Because set elements are unordered, it is not sensible to use positional subscripting. However, it is possible to iterate over *all* elements 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
```

3 Generalized Sets

There are several extensions of “ordinary” sets such as *fuzzy sets* and *multisets*. Both can be be seen as special cases of *fuzzy multisets*. For all extensions, the approach is to define a generalized set X as a pair (D, f) where D is an ordinary set representing the domain, and f the characteristic function of X, mapping D to some image I. The subset of the domain for which f is non-zero is the support of X. If $I = \{0, 1\}$, X represents an “ordinary” set. If $I = \mathbb{N}$, X becomes a multiset whose elements e_i can appear multiple times. $f(e_i)$ is then called the multiplicity of e_i. If I is the unit interval, X becomes a fuzzy set. In this context, f is typically called the membership function, $f(e_i)$ the membership grade of e_i, and D the universe for X. If I is a multiset whose domain is the unit interval (0 excluded), X is a fuzzy multiset whose elements can each have several (possibly non-unique) membership grades. If for one element, the asociated membership grades are all 1, we get a multiset. If there is at most one membership grade, we get a “simple” fuzzy set. If for the latter case the membership is 1, we fall back to an ordinary set.

Generalized sets are created using the `gset()` function. This can be done in four ways:

1. Specify the support only (this yields an ordinary set).
2. Specify support and memberships.
3. Specify support and membership function.
4. Specify a set of elements along with their individual membership grades.

Note that for efficiency reasons, `gset()` will not store elements with zero memberships grades, i.e. really expects the support and not a domain (or universe in the fuzzy world sense).

```
> X <- c("A", "B", "C")
> ## ordinary set (X is converted to a set internally).
> gset(support = X)

{A, B, C}

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

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

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

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

> ## fuzzy set using a membership function
> f <- function(x) switch(x, A = 0.1, B = 0.2, C = 1, 0)
> gset(support = X, charfun = f)

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

> ## fuzzy multiset
> ## Here, 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]}]}
```

As for ordinary sets, the usual operations such as union, intersection, and complement are available. Additionally, the sum and the difference of sets are defined, which add and subtract multiplicities:

```
> X <- gset(c("A", "B", "C"), 4:6)
> print(X)

{A [4], B [5], C [6]}

> Y <- gset(c("B", "C", "D"), 1:3)
> print(Y)

{B [1], C [2], D [3]}

> ## union vs. sum
> gset_union(X, Y)

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

> gset_sum(X, Y)

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

> ## intersection vs. difference
> gset_intersection(X, Y)

{B [1], C [2]}

> gset_difference(X, Y)

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

> ## sum and difference for fuzzy sets
> X <- gset("a", 0.3)
> Y <- gset(c("a", "b"), c(0.3, 0.4))
> gset_sum(X, Y)

{a [0.6], b [0.4]}

> gset_sum(X, Y, set("a"))

{a [1], b [0.4]}

> gset_difference(Y, X)

{b [0.4]}
```

Note that "+" and "-" can be used instead, and that for fuzzy (multi-)sets, in general, complement and difference do not yield the same result (as for crisp sets):

```
> X - Y

{}

> gset_complement(X, Y)

{a [0.3], b [0.4]}
```

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 implication. The fuzzy logic is selected by calling `fuzzy_logic()` with a character string specifying the fuzzy logic "family", and optional parameters. Available families include: `"Zadeh"` (default), `"drastic"`, `"product"`, `"Lukasiewicz"`, `"Fodor"`, `"Frank"`, `"Hamacher"`, `"Schweizer-Sklar"`, `"Yager"`, `"Dombi"`, `"Aczel-Alsina"`, and `"Sugeno-Weber"`. A call to `fuzzy_logic()` without arguments returns the currently set fuzzy logic.

```
> 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]}

> ## Zadeh-logic (default)
> 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]}

> !X

{A [0.7], B [0.5], C [0.2]}

> ## switch logic
> fuzzy_logic("Fodor")
> ## Fodor-logic
> 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]}

> !X

{A [0.7], B [0.5], C [0.2]}
```

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}
```

```
> cut(X)
```

```
{}
```

Additionally, there is a `plot()` method for fuzzy (multi-)sets that produces a barplot for the membership vector (see Figure 1):

```
> plot(X)
```

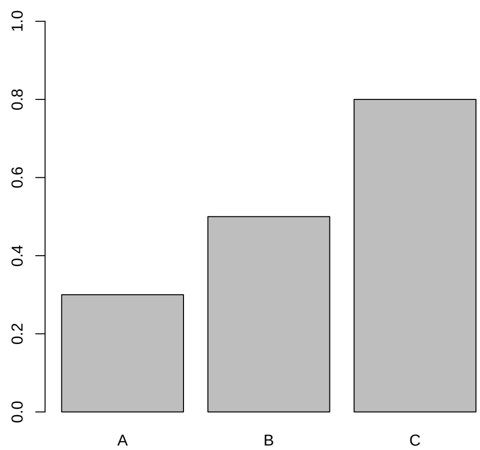

Figure 1: Membership plot for a fuzzy set.