

Package ‘ref’

February 15, 2012

Version 0.97

Date 2009-10-11

Title References for R

Author Jens Oehlschlägel <jens.oehlschlaegel@truecluster.com>

Maintainer Jens Oehlschlägel <jens.oehlschlaegel@truecluster.com>

Depends R (>= 1.5.1)

Description small package with functions for creating references, reading from and writing references and a memory efficient refdata type that transparently encapsulates matrixes and data.frames

License file LICENSE

URL <http://CRAN.r-project.org>

Encoding latin1

Repository CRAN

Date/Publication 2009-10-11 14:42:05

R topics documented:

as.ref	2
deref	2
HanoiTower	5
is.ref	8
optimal.index	9
ref	10
refdata	12

Index	16
--------------	-----------

as.ref *coercing to reference*

Description

This function RETURNS a reference to its argument.

Usage

```
as.ref(obj)
```

Arguments

obj an object existing in the current environment/frame

Value

an object of class "ref"

Author(s)

Jens Oehlschlägel

See Also

[ref](#), [deref](#)

Examples

```
v <- 1
r <- as.ref(v)
r
deref(r)
```

deref *dereferencing references*

Description

This functions allow to access a referenced object. `deref(ref)` returns the object, and `deref(ref) <- value` assigns to the referenced object.

Usage

```
deref(ref)
deref(ref) <- value
#the following does not pass R CMD CHECK
#deref<-(ref, value)
#deref(ref)[1] <- value # subsetting assignment appears to be inefficient in S+.
```

Arguments

ref	a reference as returned by ref or as.ref
value	a value to be assigned to the reference

Details

`deref` and `deref<-` provide convenient access to objects in other environments/frames. In fact they are wrappers to [get](#) and [assign](#). However, convenient does not necessarily mean efficient. If performance is an issue, the direct use of [new.env](#), [substitute](#) and [eval](#) may give better results. See the examples below.

Value

`deref` returns the referenced object.
"`deref<-`" returns a reference to the modified object, see [ref](#).

Note

Subsetting assignment appears to be inefficient in S+. Note the use of [substitute](#) in the examples.

Author(s)

Jens Oehlschlägel

References

Writing R Extensions

See Also

[ref](#), [as.ref](#), [get](#), [assign](#), [substitute](#), [eval](#)

Examples

```
# Simple usage example
x <- cbind(1:5, 1:5)          # take some object
rx <- as.ref(x)              # wrap it into a reference
deref(rx)                   # read it through the reference
deref(rx) <- rbind(1:5, 1:5) # replace the object in the reference by another one
deref(rx)[1, ]              # read part of the object
deref(rx)[1, ] <- 5:1        # replace part of the object
deref(rx)                   # see the change
```

```
cat("For examples how to pass by references see the Performance test examples at the help pages\n")
```

```
## Not run:
```

```
## Performance test examples showing actually passing by reference
```

```
# define test size
```

```
nmatrix <- 1000 # matrix size of nmatrix by nmatrix
```

```
nloop <- 10 # you might want to use less loops in S+, you might want more in R versions before 1.8
```

```
# Performance test using ref
```

```
t1 <- function(){ # outer function
```

```
  m <- matrix(nrow=nmatrix, ncol=nmatrix)
```

```
  a <- as.ref(m)
```

```
    t2(a)
```

```
  m[1,1]
```

```
}
```

```
# subsetting deref is slower (by factor 75 slower since R 1.8 compared to previous versions, and much, much slower in S+)
```

```
t2 <- function(ref){ # inner function
```

```
  cat("timing", timing.wrapper(  
    for(i in 1:nloop)
```

```
      deref(ref)[1,1] <- i
```

```
    ), "\n")
```

```
}
```

```
if (is.R())gc()
```

```
t1()
```

```
# ... than using substitute
```

```
t2 <- function(ref){
```

```
  obj <- as.name(ref$name)
```

```
  loc <- ref$loc
```

```
  cat("timing", timing.wrapper(  
    for(i in 1:nloop)
```

```
      eval(substitute(x[1,1] <- i, list(x=obj, i=i)), loc)
```

```
    ), "\n")
```

```
}
```

```
if (is.R())gc()
```

```
t1()
```

```
##
```

```
# Performance test using Object (R only)
```

```
# see Henrik Bengtsson package(oo)
```

```
Object <- function(){
```

```
  this <- list(env=new.env());
```

```
  class(this) <- "Object";
```

```
  this;
```

```
}
```

```
"$.Object" <- function(this, name){
```

```
  get(name, envir=unclass(this)$env.);
```

```
}
```

```
"$<-$.Object" <- function(this, name, value){
```

```
  assign(name, value, envir=unclass(this)$env.);
```

```
  this;
```

```
}
```

```
# outer function
```

```
t1 <- function(){
```

```

o <- Object()
o$m <- matrix(nrow=nmatrix, ncol=nmatrix)
  t2(o)
o$m[1,1]
}
# subsetting o$m is slower ...
t2 <- function(o){
  cat("timing", timing.wrapper(
    for(i in 1:nloop)
      o$m[1,1] <- i
    ), "\n")
}
if (is.R())gc()
t1()
# ... than using substitute
t2 <- function(o){
  env <- unclass(o)$env.
  cat("timing", timing.wrapper(
    for(i in 1:nloop)
      eval(substitute(m[1,1] <- i, list(i=i)), env)
    ), "\n")
}
if (is.R())gc()
t1()

## End(Not run)

```

HanoiTower

application example for references

Description

This is an example for using references in S (R/S+) with package `ref`. `HanoiTower` implements a recursive algorithm solving the Hanoi tower problem. It is implemented such that the recursion can be done either by passing the `HanoiTower` *by reference* or *by value* to the workhorse function `move.HanoiTower`. Furthermore you can choose whether recursion should use [Recall](#) or should directly call `move.HanoiTower`. As the `HanoiTower` object is not too big, it can be extended by some garbage MBytes, that will demonstrate the advantage of passing references instead of values. The deeper we recurse, the more memory we waist by passing values (and the more memory we save by passing references). Functions `move.HanoiTower` and `print.HanoiTower` are internal (not intended to be called by the user directly).

Usage

```

HanoiTower(n = 5
, parameter.mode = c("reference", "value")[1]
, recursion.mode = c("recall", "direct")[1]
, garbage = 0

```

```
, print = FALSE
, plot = TRUE
, sleep = 0
)
```

Arguments

n	number of slices
parameter.mode	one of "reference" or "value" deciding how to pass the HanoiTower object
recursion.mode	one of "recall" or "direct" deciding how to call recursively
garbage	no. of bytes to add to the HanoiTower size
print	TRUE print the HanoiTower changes
plot	FALSE not to plot the HanoiTower changes
sleep	no. of seconds to wait between HanoiTower changes for better monitoring of progress

Details

The Hanoi Tower problem can be described as follows: you have n slices of increasing size placed on one of three locations a, b, c such that the biggest slice is at the bottom, the next biggest slice on top of it and so forth with the smallest slice as the top of the tower. Your task is to move all slices from one stick to the other, but you are only allowed to move one slice at a time and you may never put a bigger slice on top of a smaller one. The recursive solution is: to move n slices from a to c you just need to do three steps: move $n-1$ slices to b , move the biggest slice to c and move $n-1$ slices from b to c . If n equals 1, just move from a to c .

Value

invisible()

Author(s)

Jens Oehlschlägel

See Also

[ref](#), [Recall](#)

Examples

```
HanoiTower(n=2)

## Not run:
# small memory examples
HanoiTowerDemoBytes <- 0
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "reference"
```

```
, recursion.mode = "direct"
, garbage = HanoiTowerDemoBytes
)
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "reference"
, recursion.mode = "recall"
, garbage = HanoiTowerDemoBytes
)
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "value"
, recursion.mode = "direct"
, garbage = HanoiTowerDemoBytes
)
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "value"
, recursion.mode = "recall"
, garbage = HanoiTowerDemoBytes
)
rm(HanoiTowerDemoBytes)

# big memory examples
HanoiTowerDemoBytes <- 100000
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "reference"
, recursion.mode = "direct"
, garbage = HanoiTowerDemoBytes
)
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "reference"
, recursion.mode = "recall"
, garbage = HanoiTowerDemoBytes
)
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "value"
, recursion.mode = "direct"
, garbage = HanoiTowerDemoBytes
)
if (is.R())
  gc()
HanoiTower(
  parameter.mode = "value"
```

```
, recursion.mode = "recall"  
, garbage = HanoiTowerDemoBytes  
)  
rm(HanoiTowerDemoBytes)  
  
## End(Not run)
```

is.ref

checking (for) references

Description

is.ref checks whether an object inherits from class "ref".
exists.ref checks whether a referenced object exists.

Usage

```
is.ref(x)  
exists.ref(ref)
```

Arguments

x	an object that might be a reference
ref	a reference as returned from ref or as.ref

Value

logical scalar

Author(s)

Jens Oehlschlägel

See Also

[ref](#), [exists](#), [inherits](#), [class](#)

Examples

```
v <- 1  
good.r <- as.ref(v)  
bad.r <- ref("NonExistingObject")  
is.ref(v)  
is.ref(good.r)  
is.ref(bad.r)  
exists.ref(good.r)  
exists.ref(bad.r)
```

optimal.index	<i>creating standardized, memory optimized index for subsetting</i>
---------------	---

Description

Function `optimal.index` converts an index specification of type {logical, integer, -integer, character} into one of {integer, -integer} whatever is smaller. Function `need.index` returns TRUE if the index does represent a subset (and thus indexing is needed). Function `posi.index` returns positive integers representing the (sub)set.

Usage

```
optimal.index(i, n=length(i.names), i.names = names(i), i.previous = NULL, strict = TRUE)
need.index(oi)
posi.index(oi)
```

Arguments

<code>i</code>	the original one-dimensional index
<code>n</code>	length of the indexed dimension (potential iMax if i where integer), not necessary if names= is given
<code>i.names</code>	if i is character then names= represents the names of the indexed dimension
<code>i.previous</code>	if i.previous= is given, the returned index represents <code>x[i.previous][i] == x[optimal.index]</code> rather than <code>x[i] == x[optimal.index]</code>
<code>strict</code>	set to FALSE to allow for NAs and duplicated index values, but see details
<code>oi</code>	a return value of <code>optimal.index</code>

Details

When `strict=TRUE` it is expected that `i` does not contain NAs and no duplicated index values. Then `identical(x[i], x[optimal.index(i, n=length(x), i.names=names(x))$i]) == TRUE`. When `strict=FALSE` `i` may contain NAs and/or duplicated index values. In this case length optimization is not performed and `optimal.index` always returns positive integers.

Value

`optimal.index` returns the index `oi` with attributes `n=n` and `ni=length(x[optimal.index])` (which is `n-length(i)` when `i` is negative). `need.index` returns a logical scalar `posi.index` returns a vector of positive integers (or `integer(0)`)

Note

`need.index(NULL)` is defined and returns FALSE. This allows a function to have an optional parameter `oi=NULL` and to determine the need of subsetting in one request.

Author(s)

Jens Oehlschlägel

See Also[refdata](#)please ignore the following unpublished links: [ids2index](#), [shift.index](#), [startstop2index](#)**Examples**

```

l <- letters
names(l) <- letters
stopifnot({i <- 1:3 ; identical(l[i], l[optimal.index(i, n=length(l))])})
stopifnot({i <- -(4:26) ; identical(l[i], l[optimal.index(i, n=length(l))])})
stopifnot({i <- c(rep(TRUE, 3), rep(FALSE, 23)) ; identical(l[i], l[optimal.index(i, n=length(l))])})
stopifnot({i <- c("a", "b", "c"); identical(l[i], l[optimal.index(i, i.names=names(l))])})
old.options <- options(show.error.messages=FALSE); stopifnot(inherits(try(optimal.index(c(1:3, 3), n=length(l)),
stopifnot({i <- c(1:3, 3, NA); identical(l[i], l[optimal.index(i, n=length(l), strict=FALSE)])})
stopifnot({i <- c(-(4:26), -26); identical(l[i], l[optimal.index(i, n=length(l), strict=FALSE)])})
stopifnot({i <- c(rep(TRUE, 3), rep(FALSE, 23), TRUE, FALSE, NA); identical(l[i], l[optimal.index(i, n=length(l),
stopifnot({i <- c("a", "b", "c", "a", NA); identical(l[i], l[optimal.index(i, i.names=names(l), strict=FALSE)])})
rm(l)

```

ref

*creating references***Description**

Package `ref` implements references for S (R/S+). Function `ref` creates references. For a memory efficient wrapper to matrixes and data.frames which allows nested subsetting see [refdata](#)

Usage

```
ref(name, loc = parent.frame())
```

Arguments

<code>name</code>	name of an (existing) object to be referenced
<code>loc</code>	location of the referenced object, i.e. an environment in R or a frame in S+

Details

In S (R/S+) paramters are passed by value and not by reference. When passing big objects, e.g. in recursive algorithms, this can quickly eat up memory. The functions of package `ref` allow to pass references in function calls. The implementation is purely S and should work in R and S+. Existence of the referenced object is not checked by function `ref`. Usually [as.ref](#) is more convenient and secure to use. There is also a print method for references.

Value

a list with

name	name of the referenced object
loc	location of the referenced object, i.e. an environment in R or a frame in S+

and class "ref"

WARNING

Usually functions in S have no side-effects except for the main effect of returning something. Working with references circumvents this programming style and can have considerable side-effects. You are using it at your own risk.

R 1.8 WARNING

Changing parts of referenced objects has been slowed down by order of magnitudes since R version 1.8, see performance test examples on the help page for [deref](#). Hopefully the old performance can be restored in future versions.

S+ WARNING

Package `ref` should generally work under R and S+. However, when changing very small parts of referenced objects, using references under S+ might be inefficient (very slow with high temporary memory requirements).

Historical remarks

This package goes back to an idea submitted April 9th 1997 and code offered on August 17th 1997 on s-news. The idea of implementing references in S triggered an intense discussion on s-news. The status reached in 1997 can be summarized as follows:

1. **advantage** passing by reference can save memory compared to passing by value
2. **disadvantage** passing by reference is more dangerous than passing by value
3. **however** the implementation is purely in S, thus rather channels existing danger than adding new danger
4. **restriction** assigning to a subsetted part of a referenced object was inefficient in S+ (was S+ version 3)

Due to the last restriction the code was never submitted as a mature library. Now in 2003 we have a stable version of R and astonishingly assigning to a subsetted part of a referenced object *can* be implemented efficient. This shows what a great job the R core developers have done. In the current version the set of functions for references was dramatically simplified, the main differences to 1997 being the following:

1. **no idempotence** `deref` and `deref<-` now are a simple function and no longer are methods. This decision was made due top performance reasons. As a consequence, `deref()` no longer is idempotent: one has to know whether an object is a reference. Function `is.ref` provides a test.

2. **no write protection** The 1997 suggestion included a write protection attribute of references, allowing for read only references and allowing for references that could only be changed by functions that know the access code. Reasons for this: there is no need for readonly references (due to copy on modify) and oop provides better mechanisms for security.
3. **no static variables** The suggestion made in 1997 did include an implementation of static variables realized as special cases of references with a naming convention which reduced the risk of name collisions in the 1997 practice of assigning to frame 0. Now R has namespaces and the oop approach of Henrik Bengtsson using environments is to be preferred over relatively global static objects.

Note

Using this type of references is fine for prototyping in a non-objectoriented programming style. For bigger projects and safer programming you should consider the approach suggested by Henrik Bengtsson at <http://www.maths.lth.se/help/R/ImplementingReferences> (announced to be released as package "oo" or "classes")

Author(s)

Jens Oehlschlägel

See Also

[as.ref](#), [deref](#), [deref<-](#), [exists.ref](#), [is.ref](#), [print.ref](#), [HanoiTower](#)

Examples

```
v <- 1
r <- ref("v")
r
deref(r)
cat("For more examples see ?deref\n")
```

refdata

subsettable reference to matrix or data.frame

Description

Function refdata creates objects of class refdata which behave not totally unlike matrices or data.frames but allow for much more memory efficient handling.

Usage

```
# -- usage for R CMD CHECK, see below for human readable version -----
refdata(x)
derefdata(x)
derefdata(x) <- value
## S3 method for class 'refdata'
```

```

x[i = NULL, j = NULL, drop = FALSE, ref = FALSE]
## S3 replacement method for class 'refdata'
x[i = NULL, j = NULL, ref = FALSE] <- value
## S3 method for class 'refdata'
dim(x)
## S3 method for class 'refdata'
dimnames(x)
## S3 method for class 'refdata'
row.names(x)
## S3 method for class 'refdata'
names(x)

# -- most important usage for human beings -----
# rd <- refdata(x)           # create reference
# derefdata(rd)             # retrieve original data
# derefdata(rd) <- value    # modify original data
# rd[]                      # get all (current) data
# rd[i, j]                  # get part of data
# rd[i, j, ref=TRUE]        # get new reference on part of data
# rd[i, j] <- value         # modify part of data (now rd is reference on local copy of the data)
# rd[i, j, ref=TRUE] <- value # modify part of original data (respecting subsetting history)
# dim(rd)                   # dim of (subsetting) data
# dimnames(rd)              # dimnames of (subsetting) data

```

Arguments

x	a matrix or data.frame or any other 2-dimensional object that has operators "[" and "[<-" defined
i	row index
j	col index
ref	FALSE by default. In subsetting: FALSE returns data, TRUE returns new refdata object. In assignments: FALSE modifies a local copy and returns a refdata object embedding it, TRUE modifies the original.
drop	FALSE by default, i.e. returned data have always a dimension attribute. TRUE drops dimension in some cases, the exact result depends on whether a matrix or data.frame is embedded
value	some value to be assigned

Details

Refdata objects store 2D-data in one environment and index information in another environment. Derived refdata objects usually share the data environment but not the index environment. The index information is stored in a standardized and memory efficient form generated by [optimal.index](#). Thus refdata objects can be copied and subsetted and even modified without duplicating the data in memory. Empty square bracket subsetting (rd[]) returns the data, square bracket subsetting (rd[i, j]) returns subsets of the data as expected.

An additional argument (`rd[i, j, ref=TRUE]`) allows to get a reference that stores the subsetting indices. Such a reference behaves transparently as if a smaller matrix/data.frame would be stored and can be subsetted again recursively. With `ref=TRUE` indices are always interpreted as row/col indices, i.e. `x[i]` and `x[cbind(i, j)]` are undefined (and raise stop errors)

Standard square bracket assignment (`rd[i, j] <- value`) creates a reference to a locally modified copy of the (potentially subsetted) data.

An additional argument (`rd[i, j, ref=TRUE] <- value`) allows to modify the original data, properly recognizing the subsetting history.

A method `dim(refdata)` returns the dim of the (indexed) data.

A `dimnames(refdata)` returns the dimnames of the (indexed) data.

Value

an object of class `refdata` (appended to class attributes of data), which is an empty list with two attributes

`dat` the environment where the data `x` and its dimension `dim` is stored

`ind` the environment where the indexes `i, j` and the effective subset size `ni, nj` is stored

Note

The `refdata` code is currently R only (not implemented for S+).

Please note the following differences to matrices and dataframes:

`x[]` you need to write `x[[]]` instead of `x` in order to get all current data

`drop=FALSE` by default `drop=FALSE` which gives consistent behaviour for matrices and data.frames.

You can use the `$-` or `[[`-operator to extract single column vectors which are granted to be of a consistent data type. However, currently `$` and `[[` are only wrappers to `[`. They might be performance tuned in later versions.

`x[i]` single index subsetting is not defined, use `x[[]][i]` instead, but beware of differences between matrices and dataframes

`x[cbind()]` matrix index subsetting is not defined, use `x[[]][cbind(i, j)]` instead

`ref=TRUE` parameter `ref` needs to be used sensibly to exploit the advantages of `refdata` objects

Author(s)

Jens Oehlschlägel

See Also

[Extract](#), [matrix](#), [data.frame](#), [optimal.index](#), [ref](#)

Examples

```

## Simple usage Example
x <- cbind(1:5, 5:1)           # take a matrix or data frame
rx <- refdata(x)              # wrap it into an refdata object
rx                             # see the autoprinting
rm(x)                         # delete original to save memory
rx[]                           # extract all data
rx[-1, ]                      # extract part of data
rx2 <- rx[-1, , ref=TRUE]     # create refdata object referencing part of data (only index, no data is duplicated)
rx2                             # compare autoprinting
rx2[]                          # extract 'all' data
rx2[-1, ]                     # extract part of (part of) data
cat("for more examples look the help pages\n")

## Not run:
# Memory saving demos
square.matrix.size <- 1000
recursion.depth.limit <- 10
non.referenced.matrix <- matrix(1:(square.matrix.size*square.matrix.size), nrow=square.matrix.size, ncol=square.matrix.size)
rownames(non.referenced.matrix) <- paste("a", seq(length=square.matrix.size), sep="")
colnames(non.referenced.matrix) <- paste("b", seq(length=square.matrix.size), sep="")
referenced.matrix <- refdata(non.referenced.matrix)
recurse.nonref <- function(m, depth.limit=10){
  x <- m[1,1] # need read access here to create local copy
  gc()
  cat("depth.limit=", depth.limit, " memory.size=", memsize.wrapper(), "\n", sep="")
  if (depth.limit)
    Recall(m[-1, -1, drop=FALSE], depth.limit=depth.limit-1)
  invisible()
}
recurse.ref <- function(m, depth.limit=10){
  x <- m[1,1] # read access, otherwise nothing happens
  gc()
  cat("depth.limit=", depth.limit, " memory.size=", memsize.wrapper(), "\n", sep="")
  if (depth.limit)
    Recall(m[-1, -1, ref=TRUE], depth.limit=depth.limit-1)
  invisible()
}
gc()
memsize.wrapper()
recurse.ref(referenced.matrix, recursion.depth.limit)
gc()
memsize.wrapper()
recurse.nonref(non.referenced.matrix, recursion.depth.limit)
gc()
memsize.wrapper()
rm(recurse.nonref, recurse.ref, non.referenced.matrix, referenced.matrix, square.matrix.size, recursion.depth.limit)

## End(Not run)
cat("for even more examples look at regression.test.refdata()\n")
regression.test.refdata() # testing correctness of refdata functionality

```

Index

- *Topic **manip**
 - optimal.index, 9
 - refdata, 12
- *Topic **programming**
 - as.ref, 2
 - deref, 2
 - HanoiTower, 5
 - is.ref, 8
 - ref, 10
 - refdata, 12
- *Topic **utilities**
 - optimal.index, 9
- [.refdata (refdata), 12
- [<-.refdata (refdata), 12
- [[.refdata (refdata), 12
- [[<-.refdata (refdata), 12
- \$.refdata (refdata), 12
- \$<-.refdata (refdata), 12

- as.ref, 2, 3, 8, 10, 12
- assign, 3

- class, 8

- data.frame, 13, 14
- deref, 2, 2, 11, 12
- deref<- (deref), 2
- deref<-, 11, 12
- derefdata (refdata), 12
- derefdata<- (refdata), 12
- dim, 14
- dim.refdata (refdata), 12
- dimnames, 14
- dimnames.refdata (refdata), 12

- eval, 3
- exists, 8
- exists.ref, 12
- exists.ref (is.ref), 8
- Extract, 14

- get, 3

- HanoiTower, 5, 12

- inherits, 8
- is.ref, 8, 11, 12

- matrix, 13, 14
- move.HanoiTower (HanoiTower), 5

- names.refdata (refdata), 12
- need.index (optimal.index), 9
- new.env, 3

- optimal.index, 9, 13, 14

- plot.HanoiTower (HanoiTower), 5
- posi.index (optimal.index), 9
- print.HanoiTower (HanoiTower), 5
- print.ref, 12
- print.ref (ref), 10
- print.refdata (refdata), 12

- Recall, 5, 6
- ref, 2, 3, 6, 8, 10, 14
- refdata, 10, 12
- row.names.refdata (refdata), 12

- substitute, 3