

Package ‘proto’

January 21, 2012

Version 0.3-9.2

Date 2011-04-27

Title Prototype object-based programming

Author Louis Kates, Thomas Petzoldt

Maintainer Gabor Grothendieck <ggrothendieck@gmail.com>

Description An object oriented system using object-based, also called prototype-based, rather than class-based object oriented ideas.

Suggests graph, Rgraphviz

License GPL-2

URL <http://r-proto.googlecode.com>

Repository CRAN

Date/Publication 2011-04-29 04:39:19

R topics documented:

proto-package	2
graph.proto	3
proto	4

Index	9
--------------	----------

Description

Object-oriented programming with the prototype model. "proto" facilitates object-oriented programming using an approach that emphasizes objects rather than classes (although it is powerful enough to readily represent classes too).

Details

"proto" allows one to write object-oriented programs using the prototype model. It is a simple system that uses objects rather than classes yet is powerful enough to readily encompass classes too. The following are sources of information on "proto":

Home page	<code>u <- "http://r-proto.googlecode.com"; browseURL(u)</code>
Overview	<code>README <- system.file("README", package = "proto"); file.show(README)</code>
Invoking a demo file	<code>demo("proto")</code>
Reference Card	<code>vignette("protoref")</code>
Tutorial	<code>vignette("proto")</code>
Prototype OO concepts document	<code>u <- "http://r-proto.googlecode.com/files/prototype_approaches.pdf"; bro</code>
News	<code>RShowDoc("NEWS", package = "proto")</code>
Wish List	<code>RShowDoc("WISHLIST", package = "proto")</code>
Thanks	<code>RShowDoc("THANKS", package = "proto")</code>
License	<code>RShowDoc("COPYING", package = "proto")</code>
Citation	<code>citation(package = "proto")</code>
This File	<code>package?proto</code>
Help file	<code>?proto</code>
graph.proto Help File	<code>?graph.proto</code>

Note

See `?graph.proto` for the solution to a common Linux installation problem with Rgraphviz that proto depends on.

Examples

```
cat("parent\n")
oop <- proto(x = 10, view = function(.) paste("this is a:", .$x))
oop$ls()
oop$view()

cat("override view in parent\n")
ooc1 <- oop$proto(view = function(.) paste("this is a: **", .$x, "****"))
ooc1$view()

cat("override x in parent\n")
```

```
ooc2 <- oop$proto(x = 20)
ooc2$view()
## Not run:
g <- graph.proto()
plot(g)

## End(Not run)
```

graph.proto

Create a graph of proto objects

Description

Creates a graph of the parent/child relationships among a set of proto objects.

Usage

```
graph.proto(e, g = new("graphNEL", edgemode = "directed"), child.to.parent = TRUE)
```

Arguments

e	A proto object or an environment whose proto objects will be graphed.
g	A graph to add the edges and nodes to. Defaults to an empty graph.
child.to.parent	If TRUE then arrows are drawn from child to parent; otherwise, from parent to child.

Details

This function is used to create a graph in the sense of the "graph" package out of the parent child relationships of proto objects. All "proto" objects in "e" and their immediate parents are placed in the graph.

The objects are labelled using their ". .Name" component. If there is no ". .Name" component then their variable name in "e" is used. In the case of a parent that is not in "e", an internally generated name will be used if the parent has no ". .Name" component. If two "proto" objects to be graphed have the same name the result is undefined.

Value

Produces an object of class "graph" that can subsequently be plotted.

Note

graph.proto makes use of the Rgraphviz package in the BioConductor repository and so Rgraphviz must be installed and loaded. On Linux one gotcha is that you may need to add the graphviz shared library, e.g. to add the directory containing the .so files, to your linker path via: export LD_LIBRARY_PATH=/path/to/graphviz/libs.

Examples

```
## Not run:
a <- proto()
b <- a$proto()
g <- graph.proto()
plot(g)
g <- graph.proto(child.to.parent = FALSE) # change arrow heads
plot(g)
g <- graph.proto(g = new("graphNEL")) # undirected
plot(g)
g <- graph.proto()
attrs <- list(node = list(fillcolor = "lightgreen"),
              edge = list(color = "cyan"),
              graph = list(rankdir = "BT"))
plot(graph.proto(), attrs = attrs) # specify plot attributes

## End(Not run)
```

 proto

Prototype object-based programming

Description

proto creates or modifies objects of the proto object oriented system.

Usage

```
proto(. = parent.env(envir), expr = {}, envir =
new.env(parent = parent.frame()), ..., funEnvir )
## S3 method for class 'list'
as.proto(x, envir, parent, all.names = FALSE, ...,
funEnvir = envir, SELECT = function(x) TRUE)
isnot.function(x)
```

Arguments

.	the parent object of the new object. May be a proto object or an environment.
expr	a series of statements enclosed in braces that define the variables and methods of the object. Empty braces, the default, may be used if there are no variables or methods to add at this time.
envir	an existing prototype object or environment into which the variables and methods defined in expr are placed. If omitted a new object is created.
funEnvir	the environment of methods passed via ... are automatically set to this environment. Normally this argument is omitted, defaulting to envir; however, one can specify FALSE to cause their environment to not be set or one can specify some other environment or proto object to which their environment is to be set.
x	a list.

parent	a prototype object or environment which is to be used as the parent of the object. If <code>envir</code> is specified then its parent is coerced to parent.
...	for <code>proto</code> these are components to be embedded in the new object. For <code>as.proto.list</code> these are arguments to pass to <code>proto</code> in the case that a new object is created.
SELECT	a function which given an object returns TRUE or FALSE such that only those for which SELECT returns TRUE are kept in the returned <code>proto</code> object.
<code>all.names</code>	only names not starting with a dot are copied unless <code>all.names</code> is TRUE.

Details

The `proto` class is defined to be a subclass of the R environment class. In particular this implies that `proto` objects have single inheritance and mutable state as all environments do. `proto` creates or modifies objects of this class. It inserts all variables and functions in `expr` and then in `...` into `envir` setting the parent to `..`. The environment of all functions inserted into the environment are set to that environment. All such functions should have the receiver object as their first argument. Conventionally this is `.` (i.e. a dot). Also `.that` and `.super` variables are added to the environment. These point to the object itself and its parent. Note that `proto` can be used as a method and overridden like any other method. This allows objects to have object-specific versions of `proto`.

`as.proto` is a generic with methods for environments, `proto` objects and lists.

`as.proto.list` copies inserts a copy of each component, `e1`, of the list `x` into the the environment or `proto` object `envir` for which `FUN(e1)` is TRUE. Also, components whose name begins with a dot, `.`, are not copied unless `all.names` is TRUE (and `FUN(e1)` is TRUE). The result is a `proto` object whose parent is `parent`. If `envir` is omitted a new object is created through a call to `proto` with `parent` and `...` as arguments. If `parent` is also omitted then the current environment is used. Note that if `parent` is a `proto` object with its own `proto` method then this call to `proto` will be overridden by that method.

The utility function `isnot.function` is provided for use with `as.proto.list` to facilitate the copying of variables only.

`$` can be used to access or set variables and methods in an object.

When `$` is used for getting variables and methods, calls of the form `obj$v` search for `v` in `obj` and if not found search upwards through the ancestors of `obj` until found unless the name `v` begins with two dots `..`. In that case no upward search is done.

If `meth` is a function then `obj$meth` is an object of class `c("instantiatedProtoMethod", "function")` which is a `proto` method with the first, i.e. `proto` slot, already filled in. It is normally used in the context of a call to a method, e.g. `obj$meth(x,y)`. There also exists `print.instantiatedProtoMethod` for printing such objects. Note that an instantiated `proto` method is not the same as a `proto` method. The first has its first slot filled in whereas the second does not. If it is desired to actually return the method as a value not in the context of a call then use the form `obj$with(meth)` or `obj[[meth]]` which are similar to `with(obj, meth)` except that the variation using `with` will search through ancestors while `[[` will not search through ancestors). The difference between `obj$meth` and `obj$with(meth)` is that in the first case `obj` implicitly provides the first argument to the call so that `obj$meth(x,y)` and `obj$with(meth)(obj,x,y)` are equivalent while in the case of `obj$with(meth)` the first argument is not automatically inserted. `$.proto` also has a three argument form. If the third argument is present then it should be a list specifying the arguments at which the instantiated method is to be evaluated.

The forms `.that$meth` and `.super$meth` are special and should only be used within methods. `.that` refers to the object in which the current method is located and `.super` refers to the parent of `.that`. In both cases the receiver object must be specified as the first argument — the receiver is not automatically inserted as with other usages of `$`.

`$` can be used to set variables and methods in an object. No ancestors are searched for the set form of `$`. If the variable is the special variable `.super` then not only is the variable set but the object's parent is set to `.super`.

A `with` method is available for proto objects.

`is.proto(p)` returns TRUE if `p` is a prototype object.

`str.proto` is provided for inspecting proto objects.

Value

`proto` and `as.proto` all return proto objects. `isnot.function` returns a logical value.

Note

`proto` methods can be used with environments but some care must be taken. These can be avoided by always using proto objects in these cases. This note discusses the pitfalls of using environments for those cases where such interfacing is needed.

Note that if `e` is an environment then `e$x` will only search for `x` in `e` and no further whereas if `e` were a proto object its ancestors will be searched as well. For example, if the parent of a proto object is an environment but not itself a proto object then `.super$x` references in the methods of that object will only look as far as the parent.

Also note that the form `e$meth(...)` when used with an environment will not automatically insert `e` as the first argument and so environments can only be used with methods by using the more verbose `e$meth(e, ...)`. Even then it is not exactly equivalent since `meth` will only be looked up in `e` but not its ancestors. To get precise equivalence write the even more verbose `with(e, meth)(e, ...)`.

If the user has a proto object `obj` which is a child of the global environment and whose methods use `.super` then `.super` will refer to an environment, not a proto object (unless the global environment is coerced to a proto object) and therefore be faced with the search situation discussed above. One solution is to create an empty root object between the global environment and `obj` like this `Root <- obj$.super <- proto(.GlobalEnv)` where `Root` is the root object. Now `.super` references will reference `Root`, which is a proto object so search will occur as expected. `proto` does not provide such a root object automatically but the user can create one easily as shown, if desired.

Although not recommended, it possible to coerce the global environment to a proto object by issuing the command `as.proto(.GlobalEnv)`. This will effectively make the global environment a proto root object but has the potential to break other software, although the authors have not actually found any software that it breaks.

See Also

[as.list](#), [names](#), [environment](#)

Examples

```

oo <- proto(expr = {x = c(10, 20, 15, 19, 17)
  location <- function(.) mean(.$x) # 1st arg is object
  rms <- function(.)
    sqrt(mean((.$x - .$location())^2))
  bias <- function(., b) .$x <- .$x + b
})

debug(oo$with(rms)) # cannot use oo$rms to pass method as a value
undebug(oo$with(rms)) # cannot use oo$rms to pass method as a value

oo2 <- oo$proto( location = function(.) median(.$x) )
oo2$rms()      # note that first argument is omitted.
oo2$ls()      # list components of oo2
oo2$as.list() # contents of oo2 as a list
oo2          # oo2 itself
oo2$parent.env() # same
oo2$parent.env()$as.list() # contents of parent of oo2
oo2$print()
oo2$ls()
oo2$str()
oo3 <- oo2
oo2$identical(oo3)
oo2$identical(oo)

# start off with Root to avoid problem cited in Note
Root <- proto()
oop <- Root$proto(a = 1, incr = function(.) .$a <- .$a+1)
ooc <- oop$proto(a = 3) # ooc is child of oop but with a=3
ooc$incr()
ooc$a      # 4

# same but proto overridden to force a to be specified
oop$proto <- function(., a) { .super$proto(., a=a) }
## Not run:
ooc2 <- oop$proto() # Error. Argument "a" is missing, with no default.

## End(Not run)
ooc2 <- oop$proto(a = 10)
ooc2$incr()
ooc2$a # 11

# use of with to eliminate having to write .$a
o2 <- proto(a = 1, incr = function(.) with(., a <- a+1))
o2c <- as.proto(o2$as.list()) # o2c is a clone of o2
o2d <- o2$proto() # o2d is a child of o2
o2$a <- 2
o2c$a # a not changed by assignment in line above
o2d$a # a is changed since a not found in o2d so found in o2

p <- proto(a = 0, incr = function(., x) .$a <- .$a + x)
pc <- p$proto(a = 100)

```

```
sapply(list(p, pc), "$.proto", "incr", list(x = 7))
```

Index

*Topic **programming**

- graph.proto, 3
- proto, 4
- proto-package, 2
- . (proto), 4
- .super (proto), 4
- .that (proto), 4
- \$.proto (proto), 4
- \$<-.proto (proto), 4

- as.list, 6
- as.proto (proto), 4

- environment, 6

- graph.proto, 3

- is.proto (proto), 4
- isnot.function (proto), 4

- name.proto (graph.proto), 3
- names, 6

- print.instantiatedProtoMethod (proto), 4
- proto, 4
- proto-package, 2

- str.proto (proto), 4
- super (proto), 4

- that (proto), 4
- this (proto), 4

- with.proto (proto), 4