Package: curry (via r-universe)

September 20, 2024

Type Package

Title Partial Function Application and Currying with %<%, %-<%, %><%, and %<!%

Version 0.1.1

Date 2016-09-28

Author Thomas Lin Pedersen

Maintainer Thomas Lin Pedersen <thomasp85@gmail.com>

Description Partial application is the process of reducing the arity of a function by fixing one or more arguments, thus creating a new function lacking the fixed arguments. The curry package provides three different ways of performing partial function application by fixing arguments from either end of the argument list (currying and tail currying) or by fixing multiple named arguments (partial application). This package provides this functionality through the %<%, %-<%, and %><% operators which allows for a programming style comparable to modern functional languages. Compared to other implementations such a purrr::partial() the operators in curry composes functions with named arguments, aiding in autocomplete etc.

License GPL (>=2)

LazyData TRUE

RoxygenNote 5.0.1

Collate 'utils.R' 'curry.R' 'defaults.R' 'partial.R' 'real_curry.R' 'scaffold.R' 'tail_curry.R'

URL https://github.com/thomasp85/curry

BugReports https://github.com/thomasp85/curry/issues

Imports utils

Repository https://thomasp85.r-universe.dev

RemoteUrl https://github.com/thomasp85/curry

RemoteRef HEAD

RemoteSha e68ed51ec823f81986774ce855da278559d79f13

Contents

`curry` *Curry a function from the start*

Description

The `curry` function and the `%<%` operator performs currying on a function by partially applying the first argument, returning a function that accepts all but the first arguments of the former function. If the first argument is `...` the curried argument will be interpreted as part of the ellipsis and the ellipsis will be retained in the returned function. It is thus possible to curry functions comtaining ellipis arguments to infinity (though not adviced).

Usage

```
fun %<% arg

curry(fun, arg)
```

Arguments

`fun`	A function to be curried. Can be any function (normal, already curried, primitives).
`arg`	The value that should be applied to the first argument.

Value

A function with the same arguments as `fun` except for the first, unless the first is `...` in which case it will be retained.

Note

Multiple currying does not result in multiple nested calls, so while the first currying adds a layer around the curried function, potentially adding a very small performance hit, currying multiple times will not add to this effect.

See Also

Other partials: `partial`, `tail_curry`

Examples

```
# Equivalent to curry(`+`, 5)
add_5 <- `+` %<% 5
add_5(10)

# ellipsis are retained when currying
bind_5 <- cbind %<% 5
bind_5(1:10)
```

`partial` *Apply arguments partially to a function*

Description

The `partial` function and the `%><%` operator allows you to partially call a function with a list of arguments. Named elements in the list will be matched to function arguments and these arguments will be removed from the returned function. Unnamed elements are only allowed for functions containing an ellipsis, in which case they are considered part of the ellipsis.

Usage

```
fun %><% args

partial(fun, args)
```

Arguments

`fun`	A function to be partially applied. Can be any function (normal, already partially applied, primitives).
`args`	A list of values that should be applied to the function.

Value

A function with the same arguments as `fun` except for the ones given in `args`

Note

Multiple partial application does not result in multiple nested calls, so while the first partial call adds a layer around the called function, potentially adding a very small performance hit, partially calling multiple times will not add to this effect.

See Also

Other partials: `curry`, `tail_curry`

Examples

```
dummy_lengths <- vapply %><% list(FUN = length, FUN.VALUE = integer(1))
test_list <- list(a = 1:5, b = 1:10)
dummy_lengths(test_list)
```

`set_defaults` *Change the defaults of a function*

Description

The `set_defaults` function and the `%<?%` operator modifies the defaults of a function, returning a new function. As such it can be thought of as a soft partial application in that the arguments does not become fixed and the arity doesn't change, but the arguments can be ignored when making the final call.

Usage

```
set_defaults(fun, defaults)

fun %<?% defaults
```

Arguments

`fun`	A function whose argument default(s) should be changed
`defaults`	A named list of values. The values will be set as default for the arguments matching their name. Non-matching elements will be ignored

Value

A new function with changed defaults

Examples

```
testfun <- function(x = 1, y = 2, z = 3) {
  x + y + z
}
testfun()

testfun2 <- testfun %<?% list(y = 10)
testfun2()
```

strict_curry *Perform strict currying of a function*

Description

The curry function and %<% operator does not perform currying in the strictest sense since it is really "just" a partial application of the first argument. Strictly speaking currying transforms a function into a function taking a single argument and returning a new function accepting a new argument as long as the original function has arguments left. Once all arguments has been consumed by function calls it evaluates the original call and returns the result. Strict currying has less applicability in R as it cannot work with functions containing '...' in its argument list as it will never reach the end of the argument list and thus never evaluate the call. Strict currying is still provided here for completeness. The Curry() function turns a function into a curried function reducing the ariety to one. The %<!% operator both transforms the function into a curried one and calls it once with the first argument. Once a function is curried it is still possible to use %<%, %-<%, and %><% though they all performs the same operation as the function does only accept a single argument. As with the other functions in the curry package, argument names and defaults are retained when performing strict currying. Calling a curried function without providing a value for it will call it with the default or set the argument to missing.

Usage

```
fun %<!% arg

Curry(fun)
```

Arguments

fun	A function to be turned into a curried function.
arg	A value to be used when calling the curried function

Value

A function accepting a single argument and returing either a new function accepting a single argument, or the result of the function call if all arguments have been provided.

Examples

```
testfun <- function(x, y, z) {
  x + y + z
}
curriedfun <- Curry(testfun)
curriedfun(1)(2)(3)

# Using the operator
testfun %<!% 1 %<!% 2 %<!% 3

# The strict operator is only required for the first call
```

```
testfun %<!% 1 %<% 2 %<% 3
```

`tail_curry` *Curry a function from the end*

Description

The `tail_curry` function and the `%-<%` operator performs currying on a function by partially applying the last argument, returning a function that accepts all but the last arguments of the former function. If the last argument is `...` the curried argument will be interpreted as the last named argument. If the only argument to the function is `...` the curried argument will be interpreted as part of the ellipsis and the ellipsis will be retained in the returned function. It is thus possible to curry functions comtaining ellipis arguments to infinity (though not adviced).

Usage

```
fun %-<% arg

tail_curry(fun, arg)
```

Arguments

`fun`	A function to be curried from the end. Can be any function (normal, already (tail_)curried, primitives).
`arg`	The value that should be applied to the last argument.

Value

A function with the same arguments as `fun` except for the last named argument, unless the only one is `...` in which case it will be retained.

Note

Multiple tail_currying does not result in multiple nested calls, so while the first tail_currying adds a layer around the curried function, potentially adding a very small performance hit, tail_currying multiple times will not add to this effect.

See Also

Other partials: `curry`, `partial`

Examples

```
# Equivalent to tail_curry(`/`, 5)
divide_by_5 <- `/` %-<% 5
divide_by_5(10)

no_factors <- data.frame %-<% FALSE
no_factors(x = letters[1:5])
```

Index