

Package ‘gestalt’

July 22, 2025

Title Tools for Making and Combining Functions

Version 0.2.0

Description Provides a suite of function-building tools centered around a (forward) composition operator, `%>>>%`, which extends the semantics of the 'magrittr' `%>%` operator and supports 'Tidyverse' quasiquotation. It enables you to construct composite functions that can be inspected and transformed as list-like objects. In conjunction with `%>>>%`, a compact function constructor, `fn()`, and a partial-application constructor, `partial()`, are also provided; both support quasiquotation.

License MIT + file LICENSE

URL <https://github.com/egnha/gestalt>

BugReports <https://github.com/egnha/gestalt/issues>

Depends R (>= 3.3.0)

Imports rlang (>= 1.0.0), utils

Suggests magrittr (>= 1.5), testthat (>= 3.0.0), knitr, rmarkdown

Collate 'gestalt.R' 'utils.R' 'closure.R' 'compose.R' 'constant.R' 'partial.R' 'fn.R' 'context.R' 'posure.R'

RoxygenNote 7.2.0

VignetteBuilder knitr

Encoding UTF-8

ByteCompile true

Config/testthat/edition 3

NeedsCompilation no

Author Eugene Ha [aut, cre]

Maintainer Eugene Ha <eha@posteo.de>

Repository CRAN

Date/Publication 2022-08-22 15:00:02 UTC

Contents

compose	2
constant	8
context	10
fn	13
partial	16
posure	18
Index	22

compose	<i>Compose Functions</i>
---------	--------------------------

Description

To compose functions,

- Use `compose()`:

```
compose(f, g, h, ...)
```

This makes the function that applies `f`, then `g`, then `h`, etc. It has the [formals](#) of the first function applied (namely `f`). For example, if

```
fun <- compose(paste, toupper)
```

then the function `fun()` has the same signature as `paste()`, and the call

```
fun(letters, collapse = ",")
```

is equivalent to the composite call

```
toupper(paste(letters, collapse = ","))
```

- Use ``%>>%``:

```
f %>>% g %>>% h %>>% ...
```

It comprehends both the semantics of the **magrittr** ``%>%`` operator and [quasiquotation](#). For example, if

```
sep <- ""
fun <- sample %>>% paste(collapse = !!sep)
```

then the function `fun()` has the same signature as `sample()`, and the call

```
fun(x, size, replace, prob)
```

is equivalent to the composite call

```
paste(sample(x, size, replace, prob), collapse = "")
```

Use [as.list\(\)](#) to recover the list of composite functions. For example, both

```
as.list(compose(paste, capitalize = toupper))
```

```
as.list(paste %>>>% capitalize: toupper)
```

return the (named) list of functions `list(paste, capitalize = toupper)`.

Usage

```
compose(...)
```

```
fst %>>>% snd
```

Arguments

...	Functions or lists thereof to compose, in order of application. Lists of functions are automatically spliced in. Unquoting of names, via <code>!!</code> on the left-hand side of <code>:=</code> , and splicing , via <code>!!!</code> , are supported.
fst, snd	Functions. These may be optionally named using a colon (<code>:</code>), e.g., <code>f %>»% nm: g</code> names the g-component "nm" (see ‘Exceptions to the Interpretation of Calls as Functions’). Quasiquote and the magrittr <code>`%>%`</code> semantics are supported (see ‘Semantics of the Composition Operator’, ‘Quasiquote’ and ‘Examples’).

Value

Function of class `CompositeFunction`, whose [formals](#) are those of the first function applied (as a closure).

Semantics of the Composition Operator

The ``%>»%`` operator adopts the semantics of the **magrittr** ``%>%`` operator:

1. **Bare names are matched to functions:** For example, in a composition like

```
... %>>>% foo %>>>% ...
```

the ‘foo’ is matched to the function of that name.

2. **Function calls are interpreted as a unary function of a point (.):** A *call* is interpreted as a *function* (of a point) in one of two ways:

- If the point matches an argument value, the call is literally interpreted as the body of the function. For example, in the compositions

```
... %>>>% foo(x, .) %>>>% ...
```

```
... %>>>% foo(x, y = .) %>>>% ...
```

the ‘foo(x, .)’, resp. ‘foo(x, y = .)’, is interpreted as the function `function(..., . = ..1) foo(x, .)`, resp. `function(..., . = ..1) foo(x, y = .)`.

- Otherwise, the call is regarded as implicitly having the point as its first argument before being interpreted as the body of the function. For example, in the compositions

```
... %>>>% foo(x) %>>>% ...
```

```
... %>>>% foo(x, y(.)) %>>>% ...
```

the 'foo(x)', resp. 'foo(x, y(.))', is interpreted as the function `function(..., . = ..1) foo(. , x)`, resp. `function(..., . = ..1) foo(. , x, y(.))`.

3. **Expressions { ... } are interpreted as a function of a point (.):** For example, in a composition

```
... %>>>% {
  foo(.)
  bar(.)
} %>>>% ...
```

the '{foo(.); bar(.)}' is interpreted as the function `function(..., . = ..1) {foo(.); bar(.)}`.

Curly braces are useful when you need to circumvent '~%>>>%~'s usual interpretation of function calls. For example, in a composition

```
... %>>>% {foo(x, y(.))} %>>>% ...
```

the '{foo(x, y(.))}' is interpreted as the function `function(..., . = ..1) foo(x, y(.))`. There is no point as first argument to foo.

Exceptions to the Interpretation of Calls as Functions: As a matter of convenience, some exceptions are made to the above interpretation of calls as functions:

- **Parenthesis ()** applies grouping. (In R, '~(' is indeed a function.) In particular, expressions within parentheses are literally interpreted.
- **Colon (:)** applies *naming*, according to the syntax '<name>: <function>', where '<function>' is interpreted according to the semantics of '~%>>>%~'. For example, in

```
... %>>>% aName: foo %>>>% ...
```

the function foo is named "aName".

- **fn()**, namespace operators ('::', ':::') and **extractors** ('\$', '[[', '[') are literally interpreted. This allows for list extractors to be applied to composite functions appearing in a '~%>>>%~' call (see 'Operate on Composite Functions as List-Like Objects'). For example, the compositions

```
paste %>>>% tolower
```

```
paste %>>>% base::tolower
```

```
(paste %>>>% toupper)[[1]] %>>>% tolower
```

are equivalent functions.

Quasiquote

The '~%>>>%~' operator supports Tidyverse [unquoting](#) (via '!'). Use it to:

- **Enforce immutability:** For example, by unquoting res in

```
res <- "result"
get_result <- identity %>>>% lapply(`[[`, !!res)
```

you ensure that the function `get_result()` always extracts the component named "result", even if the binding `res` changes its value or is removed altogether.

- **Interpret the point (.) in the lexical scope:** Even though ``%>>%`` interprets `'.'` as a function argument, you can still reference an object of that name via unquoting. For example,

```
. <- "point"
is_point <- identity %>>% {. == !!.}
```

determines a function that checks for equality with the string "point".

- **Name composite functions, programmatically:** For example, unquoting `nm` in

```
nm <- "aName"
... %>>% !!nm: foo %>>% ...
```

names the 'foo'-component of the resulting composite function "aName".

- **Accelerate functions by fixing constant dependencies:** For example, presuming the value of the call `f()` is *constant* and that `g` is a *pure* function (meaning that its return value depends only on its input), both

```
... %>>% g(f()) %>>% ...

... %>>% g(!f()) %>>% ...
```

would be functions yielding the same values. But the first would compute `f()` anew with each call, whereas the second would simply depend on a fixed, pre-computed value of `f()`.

Operate on Composite Functions as List-Like Objects

You can think of a composite function as embodying the (possibly nested) structure of its list of constituent functions. In fact, you can apply familiar index and assignment operations to a composite function, as if it were this list, getting a function in return. This enables you to leverage composite functions as *structured computations*.

Indexing: For instance, the 'sum' in the following composite function

```
f <- abs %>>% out: (log %>>% agg: sum)
```

can be [extracted](#) in the usual ways:

```
f[[2]][[2]]
f[[c(2, 2)]]

f$out$agg
f[["out"]][["agg"]]
f[["out"]]$agg

f$out[[2]]
f[[list("out", 2)]]
```

The last form of indexing with a mixed list is handy when you need to create an index programmatically.

Additionally, you can excise sub-composite functions with `[`, `head()`, `tail()`. For example:

- Both `f[1]` and `head(f, 1)` get the ‘abs’ as a composite function, namely `compose(abs)`
- `f[2:1]` reverses the order of the top-level functions to yield
`out: (log %>>>% agg: sum) %>>>% abs`
- `f$out[c(FALSE, TRUE)]` gets the ‘sum’ as a (named) composite function

Subset Assignment: Similarly, subset assignment works as it does for lists. For instance, you can replace the ‘sum’ with the identity function:

```
f[[2]][[2]] <- identity

f$out$agg <- identity
f[["out"]][["agg"]] <- identity

f$out[[2]] <- identity
f[[list("out", 2)]] <- identity
```

Multiple constituent functions can be reassigned using `[<-`. For example

```
f[2] <- list(log)

f["out"] <- list(log)

f[c(FALSE, TRUE)] <- list(log)
```

all replace the second constituent function with `log`, so that `f` becomes `abs %>>>% log`.

Other List Methods: The generic methods `unlist()`, `length()`, `names()` also apply to composite functions. In conjunction with `compose()`, you can use `unlist()` to “flatten” compositions. For example

```
compose(unlist(f, use.names = FALSE))
```

gives a function that is identical to

```
abs %>>>% log %>>>% sum
```

Composite Functions Balance Speed and Complexity

The speed of a composite function made by `compose()` or ``%>>>%`` (regardless of its nested depth) is on par with a manually constructed *serial* composition. This is because `compose()` and ``%>>>%`` are **associative**, semantically and operationally. For instance, triple compositions,

```
compose(f, g, h)
f %>>>% g %>>>% h

compose(f, compose(g, h))
f %>>>% (g %>>>% h)

compose(compose(f, g), h)
(f %>>>% g) %>>>% h
```

are all implemented as the *same function*. Lists of functions are automatically “flattened” when composed.

Nevertheless, the original nested structure of constituent functions is faithfully recovered by `as.list()`. In particular, `as.list()` and `compose()` are **mutually invertible**: `as.list(compose(fs))` is the same as `fs`, when `fs` is a (nested) list of functions. (But note that the names of the list of composite functions is always a character vector; it is never `NULL`.)

See Also

`constant()`; combined with ``%>»%``, this provides a lazy, structured alternative to the **magrittr** ``%>%`` operator.

Examples

```
# Functions are applied in the order in which they are listed
inv <- partial(`/`, 1) # reciprocal
f0 <- compose(abs, log, inv)
stopifnot(all.equal(f0(-2), 1 / log(abs(-2))))

# Alternatively, compose using the `%>»%` operator
f1 <- abs %>»% log %>»% {1 / .}
stopifnot(all.equal(f1(-2), f0(-2)))

## Not run:
# Transform a function to a JSON function
library(jsonlite)

# By composing higher-order functions:
jsonify <- {fromJSON %>»% .} %>»% {. %>»% toJSON}

# By directly composing with input/output transformers:
jsonify <- fn(f ~ fromJSON %>»% f %>»% toJSON)
## End(Not run)

# Formals of initial function are preserved
add <- function(a, b = 0) a + b
stopifnot(identical(formals(compose(add, inv)), formals(add)))

# Compositions can be provided by lists, in several equivalent ways
f2 <- compose(list(abs, log, inv))
f3 <- compose(!!! list(abs, log, inv))
f4 <- compose(abs, list(log, inv))
f5 <- compose(abs, !!! list(log, inv))
stopifnot(
  all.equal(f2, f0), all.equal(f2(-2), f0(-2)),
  all.equal(f3, f0), all.equal(f3(-2), f0(-2)),
  all.equal(f4, f0), all.equal(f4(-2), f0(-2)),
  all.equal(f5, f0), all.equal(f5(-2), f0(-2))
)

# compose() and as.list() are mutually invertible
f6 <- compose(abs, as.list(compose(log, inv)))
```

```

stopifnot(
  all.equal(f6, f0), all.equal(f6(-2), f0(-2))
)
fs <- list(abs, log, inv)
stopifnot(all.equal(check.attributes = FALSE,
  as.list(compose(fs)), fs,
))

# `>>>` supports names, magrittr `>%` semantics, and quasiquotation
sep <- ""
scramble <- shuffle: sample %>>>% paste(collapse = !!sep)
nonsense <- scramble(letters)
stopifnot(
  nchar(nonsense) == 26L,
  identical(letters, sort(strsplit(nonsense, sep)[[1]])),
  identical(scramble$shuffle, sample)
)

```

constant

Values as Functions

Description

A **constant** is a fixed value that incorporates its very computation. This is none other than a *function* that computes a fixed value when called without arguments. `constant()` declares such a function as a bona fide constant by transforming it to a function that caches the value of its void call (i.e., `constant()` **memoizes** void functions).

Combine `%>>%` with `constant()` for a *lazy, structured* alternative to the **magrittr** ``>%`` operator (see ‘Examples’).

Usage

```
constant(f)
```

```
variable(f)
```

Arguments

f	Function, or symbol or name (string) thereof, that can be called without arguments. (NB: <code>constant()</code> itself does not check whether <code>f()</code> is indeed a valid call.)
---	--

Value

`constant()` yields a function without formal arguments that returns the (cached, visibility-preserving) value of the void call `f()`.

`variable()` is the inverse transformation of `constant()`: it recovers the underlying (uncached) function of a constant function.

See Also[%>>%](#)**Examples**

```

# Function with a constant return value
val <- {message("Computing from scratch"); mtcars} %>>>%
  split(.$cyl) %>>>%
  lapply(function(data) lm(mpg ~ wt, data)) %>>>%
  lapply(summary) %>>>%
  sapply(`[[`, "r.squared")

# With every invocation, `val()` is computed anew:
val()
val()

# Declaring `val` as a constant ensures that its value is computed only once.
# On subsequent calls, the computed value is simply fetched:
const <- constant(val)
const()
const()

# As values, `val()` and `const()` are identical. But `const()`, moreover,
# has structure, namely the function `const`:
const

# For instance, you can inspect the intermediate summaries:
head(const, -1)()

# Which can itself be a constant:
summ <- constant(head(const, -1))
summ()
summ()

## Not run:
# Think of `%>>>%` combined with `constant()` as a lazy, structured
# alternative to the magrittr `%>%` operator.
library(magrittr)

val2 <- mtcars %>%
  split(.$cyl) %>%
  lapply(function(data) lm(mpg ~ wt, data)) %>%
  lapply(summary) %>%
  sapply(`[[`, "r.squared")

# `val2` and `const()` are identical values. But whereas `val2` is computed
# immediately and carries no structure, `const` embodies the process that
# produces its value, and allows you to defer its realization to the
# invocation `const()`.
stopifnot(identical(val2, const()))
## End(Not run)

```

```
# Use `variable()` to recover the original (\dQuote{variable}) function
val_var <- variable(const)
stopifnot(identical(val_var, val))
val_var()
val_var()
```

context

Run an Action in an Ordered Context

Description

Programming in R typically involves:

1. Making a context: assigning values to names.
2. Performing an action: evaluating an expression relative to a context.

`let()` and `run()` enable you to treat these procedures as reusable, *composable* components.

- `let()` makes a **context**: it *lazily* binds a sequence of ordered named expressions to a child of a given environment (by default, the current one).

For instance, in an environment `env` where `z` is in scope,

```
let(env, x = 1, y = x + 2, z = x * y * z)
```

is equivalent to calling

```
local({
  x <- 1
  y <- x + 2
  z <- x * y * z
  environment()
})
```

except `let()` binds the named expressions lazily (as [promises](#)) and comprehends tidyverse [quasiquotation](#).

- `run()` performs an **action**: it evaluates an expression relative to an environment (by default, the current one) and, optionally, a sequence of *lazily evaluated* ordered named expressions.

For instance, in an environment `env` where `x` is in scope,

```
run(env, x + y + z, y = x + 2, z = x * y * z)
```

is equivalent to calling

```
local({
  y <- x + 2
  z <- x * y * z
  x + y + z
})
```

except `run()`, like `let()`, binds `y` and `z` lazily and comprehends quasiquotation.

Usage

```
let(`_data` = parent.frame(), ...)

run(`_data` = parent.frame(), `_expr`, ...)
```

Arguments

<code>_data</code>	Context of named values, namely an environment, list or data frame; if a list or data frame, it is interpreted as an environment (like the <code>envir</code> argument of eval()).
<code>...</code>	Named expressions. An expression looks up values to the left of it, and takes precedence over those in <code>_data`</code> . Quasiquotation of names and expressions is supported (see ‘Examples’).
<code>`_expr`</code>	Expression to evaluate (“run”). Quasiquotation is supported.

Value

`run()` returns the evaluation of ``_expr`` in the combined environment of ``_data`` and `...`.

`let()` returns an environment where the bindings in `...` are in scope, as [promises](#), as if they were assigned from left to right in a child of the environment defined by ``_data``.

Composing Contexts

Contexts, as made by `let()`, have an advantage over ordinary local assignments because contexts are both lazy and composable. Like assignments, the order of named expressions in a context is significant.

For example, you can string together contexts to make larger ones:

```
foo <-
  let(a = ., b = a + 2) %>>>%
  let(c = a + b) %>>>%
  run(a + b + c)

foo(1)
#> [1] 8
```

Earlier bindings can be overridden by later ones:

```
bar <-
  foo[1:2] %>>>%           # Collect the contexts of 'foo()'
  let(c = c - 1) %>>>%     # Override 'c'
  run(a + b + c)

bar(1)
#> [1] 7
```

Bindings are [promises](#); they are only evaluated on demand:

```
run(let(x = a_big_expense(), y = "avoid a big expense"), y)
#> [1] "avoid a big expense"
```

Remark

“Contexts” as described here should not be confused with “contexts” in **R’s internal mechanism**.

See Also

`with()` is like `run()`, but more limited because it doesn’t support quasiquotation or provide a means to override local bindings.

Examples

```
# Miles-per-gallon of big cars
mtcars$mpg[mtcars$cyl == 8 & mtcars$disp > 350]
run(mtcars, mpg[cyl == 8 & disp > 350])
run(mtcars, mpg[big_cars], big_cars = cyl == 8 & disp > 350)

# 'let()' makes a reusable local context for big cars
cars <- let(mtcars, big = cyl == 8 & disp > 350)

eval(quote(mpg[big]), cars) # Quoting restricts name lookup to 'cars'
run(cars, mpg[big])        # The same, but shorter and more transparent

run(cars, wt[big])
mtcars$wt[mtcars$cyl == 8 & mtcars$disp > 350]

# Precedence of names is from right to left ("bottom-up"):
a <- 1000
run(`_expr` = a + b, a = 1, b = a + 2) # 4: all references are local
run(list(a = 1), a + b, b = a + 2)     # 4: 'b' references local 'a'
run(let(a = 1, b = a + 2), a + b)       # 4: 'b' references local 'a'
run(let(a = 1, b = a + 2), a + b, a = 0) # 3: latter 'a' takes precedence
run(list(a = 1, b = a + 2), a + b)      # 1003: 'b' references global 'a'

# Bound expressions are lazily evaluated: no error unless 'x' is referenced
run(`_expr` = "S'all good, man.", x = stop("!"))
run(let(x = stop("!")), "S'all good, man.")
let(x = stop("!")) # Environment binding 'x'
try(let(x = stop("!"))$x) # Error: !

# Quasiquotation is supported
a <- 1
run(let(a = 2), a + !!a) #> [1] 3
run(let(a = 1 + !!a, b = a), c(a, b)) #> [1] 2 2
```

Description

`fn()` enables you to create (anonymous) functions, of arbitrary call signature. Use it in place of the usual `function()` invocation whenever you want to:

- **Be concise:** The function declarations

```
fn(x, y = 1 ~ x + y)
```

```
function(x, y = 1) x + y
```

are equivalent.

- **Enforce immutability:** By enabling Tidyverse [quasiquotation](#), `fn()` allows you to “burn in” values at the point of function creation. This guards against changes in a function’s enclosing environment. (See ‘Use Unquoting to Make Robust Functions’.)

`fn_()` is a variant of `fn()` that does *not* comprehend quasiquotation. It is useful when you want unquoting (``!!``) or splicing (``!!!``) operators in the function body to be literally interpreted, rather than immediately invoked. (See ‘Quasiquotation’ for a complementary way to literally interpret unquoting and splicing operators in `fn()`.)

Usage

```
fn(..., ..env = parent.frame())
```

```
fn_(..., ..env = parent.frame())
```

Arguments

<code>...</code>	Function declaration, which supports quasiquotation .
<code>..env</code>	Environment in which to create the function (i.e., the function’s enclosing environment).

Value

A function whose enclosing environment is `..env`.

Function Declarations

A **function declaration** is an expression that specifies a function’s arguments and body, as a comma-separated expression of the form

```
arg1, arg2, ..., argN ~ body
```

or

```
arg1, arg2, ..., argN, ~body
```

(Note in the second form that the body is a one-sided formula. This distinction is relevant for argument [splicing](#), see ‘Quasiquotation’.)

- To the left of `~`, you write a conventional function-argument declaration, just as in `function(<arguments>)`: each of `arg1`, `arg2`, ..., `argN` is either a bare argument (e.g., `x` or `...`) or an argument with default value (e.g., `x = 1`).
- To the right of `~`, you write the function body, i.e., an expression of the arguments.

Quasiquotation

All parts of a function declaration support Tidyverse [quasiquotation](#):

- To unquote values (of arguments or parts of the body), use `!!`:

```
z <- 0
fn(x, y = !!z ~ x + y)
fn(x ~ x > !!z)
```

- To unquote argument names (with default value), use `:=` (definition operator):

```
arg <- "y"
fn(x, !!arg := 0 ~ x + !!as.name(arg))
```

- To splice in a (formal) list of arguments, use `!!!`:

```
# NB: Body is a one-sided formula
fn(!!!alist(x, y = 0), ~ x + y)
```

Splicing allows you to treat a complete function declaration as a unit:

```
soma <- alist(x, y = 0, ~ x + y)
fn(!!!soma)
```

- To write literal unquoting operators, use `QUQ()`, `QUQS()`, which read as “quoted unquoting,” “quoted unquote-splicing,” resp. (cf. `fn_()`):

```
library(dplyr)

my_summarise <- fn(df, ... ~ {
  groups <- quos(...)
  df %>%
    group_by(QUQS(groups)) %>%
    summarise(a = mean(a))
})
```

(Source: [Programming with dplyr](#))

Use Unquoting to Make Robust Functions

Functions in R are generally **impure**, i.e., the return value of a function will *not* in general be determined by the value of its inputs alone. This is because, by design, a function may depend on objects in its **lexical scope**, and these objects may mutate between function calls. Normally this isn't a hazard.

However, if you are working interactively and sourcing files into the global environment, or using a notebook interface like **Jupyter** or **R Notebook**, it can be tricky to ensure that you haven't unwittingly mutated an object that an earlier function depends upon.

You can use unquoting to guard against such mutations.

Example: Consider the following function:

```
a <- 1
foo <- function(x) x + a
```

What is the value of `foo(1)`? It is not necessarily 2, because the value of `a` may have changed between the *creation* of `foo()` and the *calling* of `foo(1)`:

```
foo(1) #> [1] 2
```

```
a <- 0
```

```
foo(1) #> [1] 1
```

In other words, `foo()` is impure because the value of `foo(x)` depends not only on the value of `x` but also on the *externally mutable* value of `a`.

With `fn()`, you can unquote `a` to “burn in” its value at the point of creation:

```
a <- 1
foo <- fn(x ~ x + !!a)
```

Now `foo()` is a pure function, unaffected by changes to `a` in the lexical scope:

```
foo(1) #> [1] 2
```

```
a <- 0
```

```
foo(1) #> [1] 2
```

Examples

```
fn(x ~ x + 1)
fn(x, y ~ x + y)
fn(x, y = 2 ~ x + y)
fn(x, y = 1, ... ~ log(x + y, ...))
```

```
# to specify '...' in the middle, write '... = '
fn(x, ... = , y ~ log(x + y, ...))
```

```
# use one-sided formula for constant functions or commands
fn(~ NA)
```

```

fn(~ message("!"))

# unquoting is supported (using `!!` from rlang)
zero <- 0
fn(x = !!zero ~ x > !!zero)

# formals and function bodies can also be spliced in
f <- function(x, y) x + y
g <- function(y, x, ...) x - y
frankenstein <- fn(!!!formals(f), ~ !!body(g))
stopifnot(identical(frankenstein, function(x, y) x - y))

# mixing unquoting and literal unquoting is possible
# (Assume dplyr is available, which provides group_by() and `%>%`)
summariser <- quote(mean)
my_summarise <- fn(df, ... ~ {
  groups <- quos(...)
  df %>%
    group_by(QUQS(groups)) %>%          # literal unquote-splice
    summarise(a = `!!`(summariser)(a)) # substitute `mean`
})
my_summarise

# Use fn_() with fn() as a concise way to force ("pin down") bindings
# For example, the 'x' is immutable in the function produced by call_upon():
call_upon <- fn_(x ~ fn(f ~ f(!!x)))
sapply(list(sin, cos), call_upon(0)) # [1] 0 1

# Return-value checking, as a functional transformation
enforce <- fn_(condition ~
  fn(x ~ {
    stopifnot(!!substitute(condition))
    x
  })
)
no_nan <- enforce(!is.nan(x))
log_strict <- fn(x ~ no_nan(log(x)))
log_strict(2)      # [1] 0.6931472
try(log_strict(-1)) # Error: !is.nan(x) is not TRUE

```

partial

Fix a Number of Arguments to a Function

Description

`partial()` enables **partial application**: given a function, it fixes the value of selected arguments to produce a function of the remaining arguments.

`departial()` undoes the application of `partial()` by returning the original function.

Usage

```
partial(..f, ...)
```

```
departial(..f)
```

Arguments

<code>..f</code>	Function.
<code>...</code>	Argument values of <code>..f</code> to fix, specified by name or position. Captured as quosures . Unquoting and splicing are supported (see ‘Examples’). Argument values may match the <code>...</code> argument of <code>..f</code> (if present), but only when specified by name.

Details

Even while `partial()` truncates formals, it remains compatible with functions that use [missing\(\)](#) to test whether a specified argument was supplied in a call. For example, `draw3 <- partial(sample, size = 3)` works as a function that randomly draws three elements, even though `sample()` invokes `missing(size)` and `draw3()` has the form `function(x, replace, prob) {...}`.

Because partially applied functions call the original function in an ad hoc environment, impure functions that depend on the calling context as a *value*, rather than as a lexical scope, may not be amenable to `partial()`. For example, `partial(ls, all.names = TRUE)()` is not equivalent to `ls(all.names = TRUE)`, because `ls()` inspects the calling environment to produce its value, whereas `partial(ls, all.names = TRUE)()` calls `ls(all.names = TRUE)` from an (ephemeral) evaluation environment.

Value

`partial()` returns a function whose [formals](#) are a truncation of the formals of `..f` (as a closure) by the fixed arguments. NB the original default values do not appear in the formals of a partialized function, but are nonetheless applied when the function is called.

The function `partial(..f)` is identical to `..f`.

In conformance with R’s calling convention, fixed argument values are lazy [promises](#). Moreover, when forced, they are [tidily evaluated](#). Lazy evaluation of fixed arguments can be overridden via unquoting, see ‘Examples’.

When `..f` is a partially applied function, `departial(..f)` is the (closure of) the underlying function. For ordinary (non-partially applied) functions, `departial(..f)` is identical to `..f`.

Examples

```
# Arguments can be fixed by name
draw3 <- partial(sample, size = 3)
draw3(letters)

# Arguments can be fixed by position
draw3 <- partial(sample, , 3)
draw3(letters)
```

```

# Use departial() to recover the original function
stopifnot(identical(departial(draw3), sample))

# Lazily evaluate argument values by default
# The value of 'n' is evaluated whenever rnd() is called.
rnd <- partial(runif, n = rpois(1, 5))
replicate(4, rnd(), simplify = FALSE) # variable length

# Eagerly evaluate argument values with unquoting (`!!`)
# The value of 'n' is fixed when 'rnd_eager' is created.
rnd_eager <- partial(runif, n = !!rpois(1, 5))
len <- length(rnd_eager())
reps <- replicate(4, rnd_eager(), simplify = FALSE) # constant length
stopifnot(all(vapply(reps, length, integer(1)) == len))

# Mix evaluation schemes by combining lazy evaluation with unquoting (`!!`)
# Here 'n' is lazily evaluated, while 'max' is eagerly evaluated.
rnd_mixed <- partial(runif, n = rpois(1, 5), max = !!sample(10, 1))
replicate(4, rnd_mixed(), simplify = FALSE)

# Arguments to fix can be spliced
args_eager <- list(n = rpois(1, 5), max = sample(10, 1))
rnd_eager2 <- partial(runif, !!!args_eager)
replicate(4, rnd_eager2(), simplify = FALSE)

# Use rlang::exprs() to selectively evaluate arguments to fix
args_mixed <- rlang::exprs(n = rpois(1, 5), max = !!sample(10, 1))
rnd_mixed2 <- partial(runif, !!!args_mixed)
replicate(4, rnd_mixed2(), simplify = FALSE)

# partial() truncates formals by the fixed arguments, omits default values
foo <- function(x, y = x, ..., z = "z") NULL
stopifnot(
  identical(formals(partial(foo)),
    formals(foo)),
  identical(formals(partial(foo, x = 1)),
    formals(function(y, ..., z) NULL)),
  identical(formals(partial(foo, x = 1, y = 2)),
    formals(function(..., z) NULL)),
  identical(formals(partial(foo, x = 1, y = 2, z = 3)),
    formals(function(...) NULL))
)

# Nevertheless, partial() remembers default argument values when called
f <- function(x, y = x) c(x, y)
p <- partial(f, x = 1)
stopifnot(identical(p(), c(1, 1)))

```

Description

`posure()` enables you to create *efficient* variable (i.e., parameterized) [composite functions](#).

For instance, say you have a composite function such as

```
function(..., b = 2, n) {
  (sample %>>% log(base = b) %>>% rep(n))(...)
}

# Alternatively, expressed with the magrittr %>%:
function(..., b = 2, n) {
  sample(...) %>% log(base = b) %>% rep(n)
}
```

which varies according to the values of `b` and `n`. You can express this more succinctly with `posure()`, by dropping the placeholder argument (`'...'`):

```
posure(b = 2, n ~ {
  sample %>>% log(base = b) %>>% rep(n)
})
```

This creates a function with same [formals](#) and return values.

But the `posure()` version is more efficient because it creates the composite function just *once*, rather than anew with each function call. Moreover, it is *robuster* than the functionally equivalent construction with the **magrittr** ``%>%`` because `posure()` validates the constituent functions (see ‘Examples’).

Usage

```
posure(..., ..env = parent.frame())
```

Arguments

<code>...</code>	Function declaration whose body must be a function composition expressed using <code>%>>%</code> . Quasiquotation is supported. The syntax is that of <code>fn()</code> (see ‘Function Declarations’) except that declaring <code>'...'</code> among <code>...</code> is ambiguous.
<code>..env</code>	Environment in which to create the function. (You should rarely need to set this.)

Details

`posure()` **curries** composite functions. However, the main significance of `posure()` is its efficiency, which is achieved via non-standard scoping semantics (transparent to the caller). `posure()` creates the given composite function once. When the resulting variable composite function is called, its dependencies are dynamically bound to its localized *lexical* scope, for fast lookup, then removed when the function exits. Thus a **posure** is a (parameterized) [closure](#) that is *partially dynamically scoped*. (This portmanteau is due to [Henry Stanley](#).)

Value

Function with **formals** function (... , <composite_function_dependencies>), where <composite_function_dependencies> stands for the formals captured by the dots of posure(). In particular, a call of the form

```
posure(a, b = value ~ f(a, b) %>>% g(a, b))
```

produces a function with the same formals and return values as

```
function(..., a, b = value) {
  (f(a, b) %>>% g(a, b))(...)
}
```

See Also

[%>>%, fn\(\), partial\(\)](#).

Examples

```
foo <- posure(b = 2, n ~ {
  sample %>>% log(base = b) %>>% rep(n)
})
```

```
# A posure is a composite function with dependencies:
foo
```

```
set.seed(1)
foo(2^(1:10), size = 2, n = 3)
#> [1] 3 4 3 4 3 4
```

```
set.seed(1)
rep(log(sample(2^(1:10), size = 2), base = 2), 3)
#> [1] 3 4 3 4 3 4
```

```
# However, a 'posure()' does the composition upfront, so it is faster
# than the equivalent function defined using the magrittr pipe:
```

```
library(magrittr) # Provides the pipe %>%
```

```
foo_pipe <- function(..., b = 2, n) {
  sample(...) %>% log(base = b) %>% rep(n)
}
```

```
set.seed(1)
foo_pipe(2^(1:10), size = 2, n = 3)
#> [1] 3 4 3 4 3 4
```

```
# Moreover, posures are safer than functions defined using the pipe,
# because '%>>%' validates constituent functions:
try(posure(b = 2, n ~ log(Base = b) %>>% rep(n)))
# Error: unused argument (Base = b)
```

```
try(posure(b = 2 ~ my_sample %>>% log(base = b)))  
# Error: object 'my_sample' not found
```

Index

[, [5](#)
%>% (compose), [2](#)
%>%%, [8](#), [9](#), [19](#), [20](#)

as.list(), [2](#), [7](#)

closure, [19](#)
compose, [2](#)
composite functions, [19](#)
constant, [8](#)
constant(), [7](#)
context, [10](#)

departial (partial), [16](#)

enclosing environment, [13](#)
eval(), [11](#)
extracted, [5](#)
extractors, [4](#)

fn, [13](#)
fn(), [4](#), [19](#), [20](#)
fn_ (fn), [13](#)
formals, [2](#), [3](#), [17](#), [19](#), [20](#)
function(), [13](#)

head(), [5](#)

length(), [6](#)
let (context), [10](#)

missing(), [17](#)

names(), [6](#)

partial, [16](#)
partial(), [20](#)
posure, [18](#)
promises, [10](#), [11](#), [17](#)

Quasiquotation, [3](#), [11](#), [19](#)
quasiquotation, [2](#), [10](#), [13](#), [14](#)

quosures, [17](#)

run (context), [10](#)

splicing, [3](#), [14](#), [17](#)

tail(), [5](#)
tidily evaluated, [17](#)

unlist(), [6](#)
Unquoting, [3](#), [17](#)
unquoting, [4](#)

variable (constant), [8](#)

with(), [12](#)