

Dot-Pipe: an S3 Extensible Pipe with R Expression Semantics

by John Mount and Nina Zumel

Abstract Pipe notation is popular with a large league of R users, with **magrittr** being the dominant realization. However, this should not be enough to consider piping in R as a completely settled topic that is not subject to further discussion, experiments, or the possibility of improvement. To promote innovation opportunities we describe **wrpr** “dot-pipe”, a well behaved sequencing operator with R expression semantics and S3 extensibility. In this article we include a number of examples of using this pipe to interact with and extend other R packages.

Introduction

Using pipes to sequence operations has a number of advantages. Piping is analogous to representing function composition as a left to right flow of values, which is a natural direction for western readers, and is much more legible than composition represented as nesting.

Pipe notation is a popular topic in the R community. Related work includes:

- magrittr** Bache and Wickham (2014) the very popular pipe used in **dplyr** (Wickham et al., 2017).
- future** Bengtsson (2017) a powerful distributed processing package with pipe notation.
- rmonad** Arendsee (2017) a monadic operator package, capturing exceptions in addition to managing composition and values.
- pipeR** Ren (2016) a collection of sequencing methods including pipes and method chaining.
- backpipe** Brown (2016) a right to left pipe operator.
- drake** Landau (2018) A work-flow/graph toolkit for reproducible code and high-performance computing.

This article will discuss using **wrpr** (Mount and Zumel, 2018) “dot-pipe” both in user code and in packages. dot-pipe has R expression based semantics, is compatible with many other meta-programming paradigms, and is S3 extensible.

Pipe notations

In and out of R there are a number of common pipe notations:

- Mathematical function composition or application: One can write “ $a \circ b$ ” to denote “ $b(a)$ ”.
- **magrittr** pipe: “`a %>% b(...)`” most commonly is used to denote “[`. <- a; b(., ...)`]¹”. The square braces indicate the dot assignment is happening in a temporary environment that **magrittr** introduces to reduce visible side-effects.
- F#’s forward pipe operator often defined as “`a |> b`” means `b a` (uses F#’s partial application feature).
- The dot-pipe (topic of this article) where “`a %.>% b`” is intended to mean “[`. <- a; b`]”. Dot-pipe emphasizes sequencing expressions, not function composition or introducing function arguments.

Using `%.>%` to sequence operations

In this section, we demonstrate the use of **wrpr** “dot-pipe” `%.>%` and some of its merits.

The intended semantics of `%.>%` are:

“`a %.>% b`” is nearly equivalent to “[`. <- a; b`]

Where `a` and `b` are taken to be R expressions, presumably with “.” occurring as a unbound (or free) symbol in `b`.

For example:

¹To observe the use of dot in **magrittr** one can try the example “`5 %>% (function(x) substitute(x))`”.

```
library("wrapr")
5 %>% sin(.)

[1] -0.9589243

print(.)

[1] 5
```

Notice the **wrapr** dot-pipe leaves the most recent left-hand side value in the variable named `"."`. While this is a visible side-effect of this pipe which can conflict with other uses of `"."`, we feel these explicit semantics are sensible, easy to teach, and easy to work with.

We can also write `"5 %>% sin"`, as the dot-pipe looks up functions by name as a user convenience. This function lookup is a non referentially transparent special case, as names are deliberately treated differently than values. However it is an important capability that we will discuss later and greatly expand using R S3 object oriented dispatch. Dot-pipe will not work with the expression `"5 %>% sin()"` (which does work with **magrittr**) because dot-pipe treats `"sin()"` as an expression (which throws an error), and not as a function. Maintaining an explicit distinction between `"sin"` (the name of a function) and `"sin()"` (an expression) will be shown to have benefits, which we will demonstrate in the "Extending the sequencer" section. In general, for dot-pipe the explicit expression `"sin(.)"` is preferred to `"sin"` under the rubric "dot-pipe has lots of dots."

Additional dot-pipe examples include:

```
5 %>% {1 + .}

[1] 6

5 %>% (1 + .)

[1] 6
```

Notice dot-pipe treated the last two statements similarly. We warn the reader that in R the expression `"5 %>% 1 + ."` is read as `"(5 %>% 1) + ."`, as special operators (those using `"%"`) have higher operator precedence than binary arithmetic operators [R Core Team \(2018\)](#).

The dot-pipe works well with many packages, including **dplyr** (example taken from a known **dplyr**/**magrittr** incompatibility: [various \(2018a\)](#)):

```
library("dplyr")
disp <- 4
mtcars %>% filter(., .data$cyl == .env$disp) %>% nrow(.)

[1] 11
```

Extending the sequencer

Dot-pipe's primary dispatch is user extensible. As we said, it treats `"a %>% b"` as `"{. <- a; b}"`. However, it does this through S3 dispatch through a method of the form `"pipe_step(a,b,pipe_environment,pipe_name)"`. User or package code can override this method to add custom effects. For example one can extend dot-pipe to be a **ggplot2** layer compositor as we show below.²

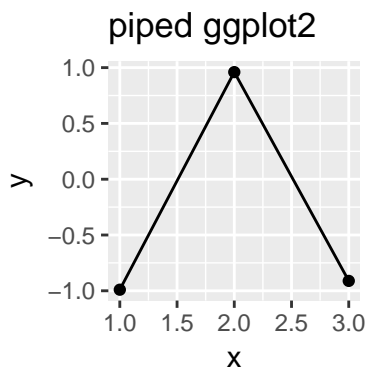
```
library("ggplot2")
pipe_step.gg <- function(pipe_left_arg,
                          pipe_right_arg,
                          pipe_environment,
                          pipe_name) {
  pipe_right_arg <- eval(pipe_right_arg,
                        envir = pipe_environment,
                        enclos = pipe_environment)
  pipe_left_arg + pipe_right_arg
}
d <- data.frame(x = 1:3, y = 11:13)
```

²For context please see [various \(2018b\)](#).

We have defined an implementation of `pipe_step.gg`, as this is the class used by `ggplot2` to recognize its own objects (i.e., `ggplot2` works by defining ``+`.gg`). Essentially `pipe_step.gg(a,b)` is implemented as `"a + b"`, the only detail being `"b"` is passed as a un-evaluated language argument, so it must be evaluated before being used as a regular value (a detail discussed in the package documentation).

We can now easily write a pipeline that combines sequencing `dplyr` transformation steps and combining `ggplot2` geom objects.

```
d %>%
  mutate(., y = cos(3*x)) %>%
  ggplot(., aes(x = x, y = y)) %>%
  geom_point() %>%
  geom_line() %>%
  ggtitle("piped ggplot2")
```



As before the data processing steps (e.g. `mutate()`) require `"."` as a free symbol to specify where the piped values go. However, the `ggplot2` steps do not use such a `"."` argument, as these functions do not expect previous steps as arguments.

Dot-pipe was able add capabilities (S3 based piping) to the `ggplot2` package without requiring any changes to the `ggplot2` package. This extension capability is important.

Explaining the plotting pipeline

The last example is easier to comprehend if we step through the translated pipeline steps in detail. For the single known sequence of realized classes from the previous example we can interpret the dot-pipe pipeline as the following (we are being slightly informal as to when each assignment occurs).

```
. <- d
. <- mutate(., y = cos(3*x))      # S3 selected pipe_step.default()
. <- ggplot(., aes(x = x, y = y)) # S3 selected pipe_step.default()
. <- . + geom_point()             # S3 selected pipe_step.gg()
. <- . + geom_line()              # S3 selected pipe_step.gg()
. + ggtitle("piped ggplot2")    # S3 selected pipe_step.gg()
```

For each of the above lines we have simulated dot-pipe's S3 dispatch during the translation by using the the known class of the value stored in `"."` to decide how to re-write the portion of the line after the assignment.

Treating names as functions

If an object on the right hand side of a dot-pipe stage is an R language name, then the object referred to by that name can be applied as a function (if it refers to a function), or declare a surrogate function (via S3 dispatch on the class of the *second* or right hand side argument) to be applied as a function. That is a `%>% b` is treated as `b(a)` or `f_class(b)(a,b)`. Notice `f` is chosen by S3 dispatch based on the run-time class of the second or right hand side argument `"b"`.

A good example use of this capability is extending the `rquery` package (Mount, 2018) to allow relational operator trees (from that package) to be used both as inspectable objects and as functions that can be applied directly to data. In the following example, we create an operator tree that adds the column `y` to a data frame `d`.

```
library("rquery")
optree <- table_source("d", colnames(d)) %>%
  extend_nse(., y = cos(2*x))
```

We can treat optree as an object as we show below.

```
class(optree)

[1] "relop_extend" "relop"

print(optree)

[1] "table('d') %>% extend(., y := cos(2 * x))"

column_names(optree)

[1] "x" "y"

columns_used(optree)

$d
[1] "x"
```

Or we can pipe into it as if it were an expression or function, as we now demonstrate.

```
wrapr_function.relop <-
  function(pipe_left_arg, pipe_right_arg,
           pipe_environment, pipe_name) {
    rquery_apply_to_data_frame(pipe_left_arg,
                               pipe_right_arg,
                               pipe_environment)
  }
d %>% optree # apply optree to d as if optree were a function

  x      y
1 1 -0.4161468
2 2 -0.6536436
3 3  0.9601703
```

In this example we defined the function stand-in for the right hand side pipe argument. To be beastly about it: wrapr_function “verbifies nouns.”³ Any user or package can extend the dot-pipe to suit their needs, just as we have shown here. If **rquery** were to include a function such as `wrapr_function.relop` then dot-pipe users would see this benefit with no extra effort on their own part. We strongly encourage package developers to start including a few “extra effects” for **wrapr** users.

Dot-pipe semantics

We have been describing dot-pipe semantics by introducing transformed code that we ask to be considered equivalent to the dot-pipe pipeline. Think of that as the specification. Dot-pipe’s implementation is not by code substitution but through execution of rules we outline here.

In R, special operators (those written with %) are left to right associative (meaning “a %>% b %>% c” is taken to mean “(a %>% b) %>% c”) with fairly high operator precedence (meaning they are applied earlier than some other operators).

The dot-pipe semantics are realized by the following processing rules. “a %>% b” is processed as follows:

1. The first or left hand side argument “a” is evaluated, with the result assigned to “.”.
2. If the second or right hand side argument “b” is not an R language name, it is considered to be a general R language expression. S3 dispatch is performed on `pipe_step(a, b, env, nm)`, with “a” being the class determining argument, and “b” an un-evaluated R language object. The default implementation of `pipe_step()` is “eval(b)”. This is the primary case for the pipe and (if no additional S3 implementations of `pipe_step()` are specified) and yields the claimed “a %>% b” is treated as “{. <- a; b}” semantics.

³This is a joke for English speakers: “practically any noun can be verbed in English” [Peterson \(1984\)](#).

3. If the second or right hand side argument “b” is a name, the name is dereferenced. This is the “function or value on the right” case which works as follows:
 - (a) If the second or right hand side argument “b” is now a function, the value `b(a)` is returned.
 - (b) Otherwise S3 dispatch is performed on `wrpr_function(a,b,env,nm)` using `b` as the class determining argument (`env` being the environment we want to be working in). We call this “class(b) choosing a surrogate function.”

This may seem involved, but it is in fact quite regular with only one exception: names on the right trigger a dereference and right-dispatch. Roughly the rule is: “treat the second or right hand side argument as an expression, unless it is a name.” The intent is for dot-pipe to have simple semantics that are capable of being combined many ways to allow rich emergent behavior.

Example Applications

Both R users and package developers can achieve a great number of useful effects by adding S3 implementations for `pipe_step()` or for `wrpr_function()`. Some possibilities include:

- Enabling `%>%` as layering function for **ggplot2** (as a replacement for “+”, as we demonstrated).
- Enabling auto-application of **rquery** operation trees to “data.frame”s (as we demonstrated).
- Enabling auto-application of models by mapping `wrpr_function.model_class` to the appropriate predict method.

```
d <- data.frame(x=1:5, y = c(1, 1, 0, 1, 0))
model <- glm(y~x, family = binomial, data = d)
wrpr_function.glm <- function(pipe_left_arg,
                              pipe_right_arg,
                              pipe_environment,
                              pipe_name) {
  predict(pipe_right_arg,
    newdata = pipe_left_arg,
    type = 'response')
}
d %>% model
```

	1	2	3	4	5
	0.9428669	0.8472400	0.6508301	0.3851524	0.1739107

Notice we can pipe new data directly into the model for prediction. The S3 `wrpr_function` extensions give us a good opportunity to regularize model predictions functions to take the same arguments and have the desired default behaviors.

- Enabling pipe notation for SQL.

```
db = DBI::dbConnect(RSQLite::SQLite(),
  "memory:")
wrpr_function.SQLiteConnection <- function(pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  pipe_name) {
  DBI::dbGetQuery(pipe_right_arg, pipe_left_arg)
}
"SELECT * FROM sqlite_temp_master" %>% db
```

```
[1] type      name      tbl_name rootpage sql
<0 rows> (or 0-length row.names)
```

```
DBI::dbDisconnect(db)
```

Here we piped SQL code directly into the database connection.

- A string concatenation operator.

```
pipe_step.character <- function(pipe_left_arg,
  pipe_right_arg,
  pipe_environment,
  pipe_name) {
```

```

    pipe_right_arg <- eval(pipe_right_arg,
                          envir = pipe_environment,
                          enclos = pipe_environment)
    paste0(pipe_left_arg, pipe_right_arg)
  }
  `+%` <- wrapr::`%>%`
  "a" %+% "b" %+% "c"

[1] "abc"

```

One can, of course, define a string concatenation operator directly- but this is a good example of the use of the dot-pipe as a sort of compound constructor.

- A formula term collector.

```

pipe_step.formula <- function(pipe_left_arg,
                              pipe_right_arg,
                              pipe_environment,
                              pipe_name) {
  pipe_right_arg <- eval(pipe_right_arg,
                        envir = pipe_environment,
                        enclos = pipe_environment)
  pipe_right_arg <- paste(pipe_right_arg, collapse = " + ")
  update(pipe_left_arg, paste(" ~ . +", pipe_right_arg))
}
`+%` <- wrapr::`%>%`
(y~a) %+% c("b", "c", "d") %+% "e"

y ~ a + b + c + d + e

```

We anticipate motivated package authors can find many special cases that the dot-pipe can streamline for their users. The value will be when many packages add effects on the same pipe, so users know by using that pipe they will simultaneously have many powerful features made available.

We have found it profitable to roughly think of `pipe_step()` as a “programmable comma” and `wrapr_function()` as “automatic execution.”⁴

Conclusion

We have demonstrated a well-behaved, S3 extensible tool for sequencing or pipe-lining operations in R. The left-dispatch of `pipe_step()` method is useful in assembling composite structures such as building a `ggplot2` plot up from pieces. The right-dispatch `wrapr_function()` is unusual, but a natural extension of the “pipes write functions on the right” idea. The goal of dot-pipe is to supply simple semantics (in the R style) that can later be composed into powerful specific applications. The dot-pipe can be used to extend packages, or to add user desired effects. We would like **wrapr** dot-pipe to be a testing ground both for pipe-aware package extensions and for experimenting with the nature of piping in R itself.

Bibliography

- Z. Arendsee. *rmonad: A Monadic Pipeline System*, 2017. URL <https://CRAN.R-project.org/package=rmonad>. R package version 0.4.0. [p1]
- S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL <https://CRAN.R-project.org/package=magrittr>. R package version 1.5. [p1]
- H. Bengtsson. *future: Unified Parallel and Distributed Processing in R for Everyone*, 2017. URL <https://CRAN.R-project.org/package=future>. R package version 1.6.2. [p1]
- C. Brown. *backpipe: Backward Pipe Operator*, 2016. URL <https://CRAN.R-project.org/package=backpipe>. R package version 0.1.8.1. [p1]
- W. M. Landau. *drake: Data Frames in R for Make*, 2018. URL <https://CRAN.R-project.org/package=drake>. R package version 5.0.0. [p1]

⁴Obviously these are vague terms, but they capture some sliver of the differences in semantics of the seen between the two patterns.

- J. Mount. *rquery: Relational Query Generator for Data Manipulation*, 2018. URL <https://CRAN.R-project.org/package=rquery>. R package version 0.3.0. [p3]
- J. Mount and N. Zumel. *wrapr: Wrap R Functions for Debugging and Parametric Programming*, 2018. <https://github.com/WinVector/wrapr>, <http://winvector.github.io/wrapr/>. [p1]
- P. L. Peterson. Semantic indeterminacy and scientific underdetermination. *Philosophy of Science*, 51(3): 464–487, September 1984. [p4]
- R Core Team. R language definition, 2018. URL <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>. [p2]
- K. Ren. *pipeR: Multi-Paradigm Pipeline Implementation*, 2016. URL <https://CRAN.R-project.org/package=pipeR>. R package version 0.6.1.3. [p1]
- various. dplyr issue 3286: .env pronoun error, only when piped, 2018a. URL <https://github.com/tidyverse/dplyr/issues/3286>. [p2]
- various. Why can't ggplot2 use %>%?, 2018b. URL <https://community.rstudio.com/t/why-cant-ggplot2-use/4372>. [p2]
- H. Wickham, R. Francois, L. Henry, and K. Müller. *dplyr: A Grammar of Data Manipulation*, 2017. URL <https://CRAN.R-project.org/package=dplyr>. R package version 0.7.4. [p1]

John Mount
Win-Vector LLC
552 Melrose Ave., San Francisco CA, 94127
USA
jmount@win-vector.com

Nina Zumel
Win-Vector LLC
552 Melrose Ave., San Francisco CA, 94127
USA
nzumel@win-vector.com