The Pipe Equivalence Myth: When `f() |> g()` Is Not the Same as `g(f())`

Ronald G. Thomas

2026-01-31

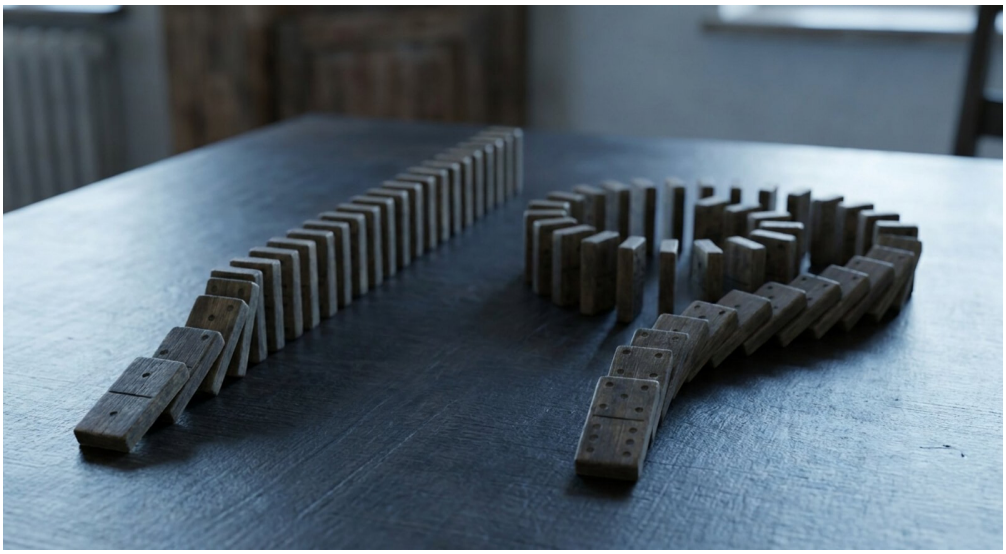

Figure 1: Two wooden domino chains side by side, one laid straight and one coiled back on itself, both ending at the same final domino: two syntactically different call structures that only sometimes reach the same result.

When pipes meet metaprogramming, the results may be surprising.

Introduction

Are `f() |> g()` and `g(f())` truly interchangeable notations? A plot wrapper that captured user expressions for logging revealed otherwise. The wrapper worked perfectly with nested calls but silently produced wrong output when piped. That experience leads directly into R's evaluation model, revealing that the "equivalence" every tutorial teaches has a significant caveat that is rarely discussed.

Most R programmers internalize the pipe as pure syntactic sugar. The documentation and textbooks reinforce this mental model, and for the vast majority of everyday code, it holds. But there is

a class of functions where the assumption breaks down completely, and understanding why reveals something fundamental about how R works under the hood.

We document the interaction between pipes, lazy evaluation, and non-standard evaluation (NSE), walking through the problem, showing exactly when and why the equivalence fails, and presenting practical workarounds for designing pipe-safe APIs.

Motivations

- To understand a silent bug in a plot wrapper function that only manifested when users piped input instead of nesting calls.
- To develop a deeper understanding of R's evaluation semantics beyond the surface-level "pipe is just sugar" explanation.
- To document a limitation that tutorials and textbooks rarely mention.
- To determine which utility function patterns using `substitute()` and `match.call()` are pipe-safe and which are not.
- To understand the relationship between referential transparency, lazy evaluation, and non-standard evaluation in R.

Objectives

1. Demonstrate that `f() |> g()` and `g(f())` produce different results when `g()` uses non-standard evaluation.
2. Explain the evaluation order difference between piped and nested calls, with step-by-step tracing.
3. Catalog which R functions and metaprogramming tools are affected by this problem.
4. Present four practical design patterns for writing functions that work correctly regardless of how they are called.

Errors and better approaches are welcome; see the Feedback section at the end.

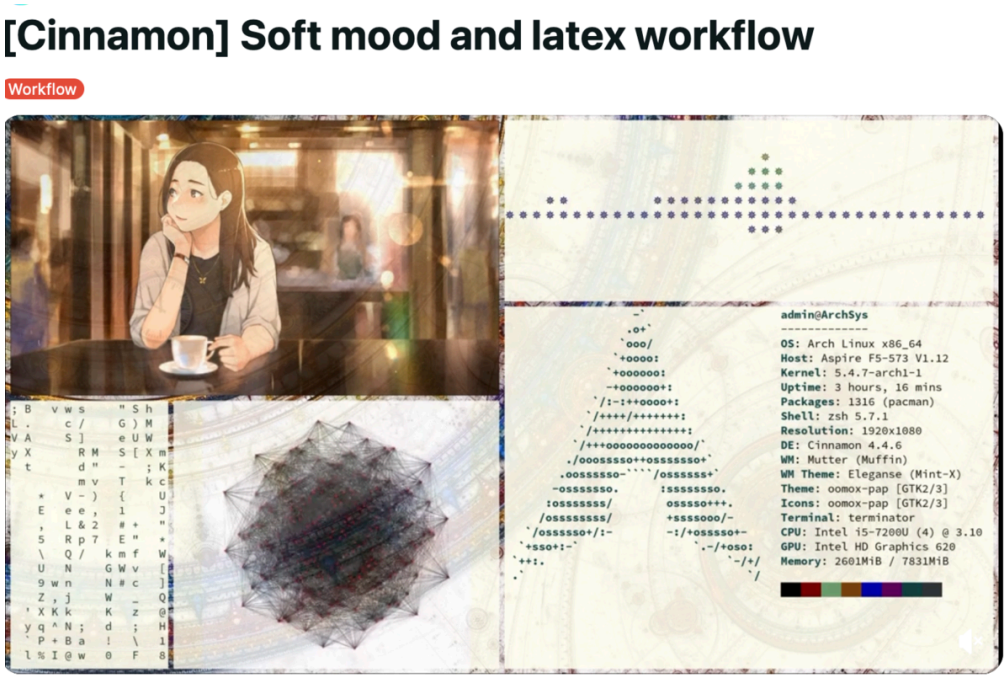

Figure 2: A moment of focus before diving into evaluation semantics.

Prerequisites and Setup

This post is conceptual and uses minimal code. No special packages are required beyond base R. The rlang package appears in one example to illustrate tidy evaluation, but the core concepts rely entirely on base R functions.

Background assumed: Familiarity with writing R functions, basic understanding of the pipe operator (`|>` or `%>%`), and awareness that R has both standard and non-standard evaluation.

```
# The only external package referenced (one example)
# install.packages("rlang")
library(rlang)
```

What is Pipe Equivalence?

Pipe equivalence is the widely-taught principle that `x |> f()` means exactly the same thing as `f(x)`. The pipe takes the result on the left and passes it as the first argument to the function on the right. Every R tutorial, textbook, and vignette states this as fact, and for value-based functions it is true. The equivalence breaks, however, when the receiving function inspects the *expression* it was given rather than just the *value*. This happens with any function that uses non-standard evaluation (functions like `substitute()`, `match.call()`, and their tidy evaluation counterparts).

The Assumption Everyone Teaches

Ask any R programmer what `x |> f()` means, and the answer will be that it is equivalent to `f(x)`. The pipe operator is just syntactic sugar for function composition, right?

This is exactly what the documentation and tutorials tell us:

> "`x %>% f` is equivalent to `f(x)`"
>
> "`x %>% f(y)` is equivalent to `f(x, y)`"
>
> "`x %>% f %>% g %>% h` is equivalent to `h(g(f(x)))`"
>
> — A popular R package vignette

And from a widely-used data science textbook:

> "The pipe takes the thing on its left and passes it along to the function on its right so that `x |> f(y)` is equivalent to `f(x, y)`, and `x |> f(y) |> g(z)` is equivalent to `g(f(x, y), z)`."

Tutorials reinforce this message:

> "Multiple pipes can be chained together, such that `x %>% f() %>% g() %>% h()` is equivalent to `h(g(f(x)))`."
>
> — A popular online R tutorial

```
# These are "the same"
sqrt(16) |> log()
log(sqrt(16))
```

Both return 1.3862944. So far, so good.

This equivalence holds for the vast majority of R code. But there is a class of functions where this assumption breaks down completely, and understanding why reveals something fundamental about how R works.

When the Equivalence Breaks

Consider a function that needs to capture the *expression* passed to it, not just its *value*:

```
capture_expr <- function(expr) {
  deparse(substitute(expr))
}

# Direct call - captures the expression
capture_expr(sqrt(16))
```

```
[1] "sqrt(16)"
```

```r
# Piped call - captures... what?
sqrt(16) |> capture_expr()
```

```
[1] "sqrt(16)"
```

The direct call captures `"sqrt(16)"` as a string. The piped version captures something entirely different – the intermediate result after `sqrt(16)` has already been evaluated.

This is not a bug. It is a fundamental consequence of how pipes work.

The Evaluation Order Problem

The pipe operator evaluates left-to-right *before* passing to the next function. By the time `capture_expr()` can call `substitute()`, it only sees the return value, not the original expression.

Here is the sequence of events:

Direct call: `capture_expr(sqrt(16))`

1. R sees the call to `capture_expr()`
2. The argument `sqrt(16)` is passed *unevaluated* (lazy evaluation)
3. Inside `capture_expr()`, `substitute(expr)` captures `sqrt(16)`
4. `deparse()` converts it to the string `"sqrt(16)"`

Piped call: `sqrt(16) |> capture_expr()`

1. R evaluates `sqrt(16)`, which returns `4`
2. The value `4` is passed to `capture_expr()`
3. Inside `capture_expr()`, `substitute(expr)` captures `4`
4. `deparse()` converts it to the string `"4"`

The critical difference: with direct calls, R's lazy evaluation means arguments are passed as *promises* containing the unevaluated expression. The pipe forces evaluation before the handoff.

Figure 3: Diving deeper into R's evaluation semantics.

Understanding the internals helps explain the surface behavior.

A Real-World Example

This is not just academic. The problem surfaces immediately when building practical tools. Consider building a plot wrapper that captures what the user typed for logging or history:

```
# Desired behavior: capture "plot(mtcars$wt,
# mtcars$mpg)" for the log
zzplot <- function(expr) {
  # Capture what the user typed
  code <- deparse(substitute(expr))

  # Open device, evaluate, close device
  png("plot.png")
  eval(
    substitute(expr),
    envir = parent.frame()
  )
  dev.off()

  # Log the code
  message("Rendered: ", code)
}
```

```r
# This works - captures the expression
zzplot(plot(mtcars$wt, mtcars$mpg))
# Message: Rendered: plot(mtcars$wt, mtcars$mpg)

# This fails - plot already executed, returns NULL
plot(mtcars$wt, mtcars$mpg) |> zzplot()
# Message: Rendered: NULL
```

The wrapper function needs to:

1. Capture the plotting code as an expression
2. Evaluate it inside a graphics device context
3. Log what was executed

With piping, step 1 fails because the plot has already been drawn to whatever device was active *before* `zzplot()` even runs.

Functions Affected by This Problem

Any function using non-standard evaluation (NSE) is potentially affected:

Function	Purpose	Pipe-safe?
`substitute()`	Capture unevaluated expression	No
`deparse(substitute())`	Convert expression to string	No
`match.call()`	Capture the entire function call	No
`enquo()` / `enquos()`	Tidy eval expression capture	No
`rlang::enexpr()`	Capture single expression	No
`quote()`	Quote an expression	Yes*
`bquote()`	Quote with substitution	Yes*

* These quote the literal argument, so piping changes *what* gets quoted.

Many tidyverse functions use NSE internally but are designed to work with pipes because they evaluate captured expressions in data contexts. However, when one writes functions that need to capture user expressions, the pipe becomes problematic.

The Native Pipe vs. Magrittr

Does the choice of pipe operator matter? Not for this problem.

Both `|>` (native, R 4.1+) and `%>%` (magrittr) evaluate the left-hand side before passing it to the right. The native pipe is essentially a syntax transformation at parse time (Wickham and Henry 2023):

```r
# Native pipe: parsed as
x |> f(y)
# becomes
f(x, y)

# But x is still evaluated before f() sees it
```

The magrittr pipe does more work behind the scenes but has the same fundamental behavior: eager evaluation of the left-hand side (Bache and Wickham 2022).

Magrittr does offer `%!>%`, an "eager pipe" that forces evaluation at each step, but this is for controlling *when* side effects occur, not for enabling expression capture.

Workarounds and Design Patterns

Pattern 1: Accept the Limitation

Designing the API to require wrapping, not piping, is the most explicit path:

```r
# Document that this is the correct usage
zzplot(
  boxplot(mpg ~ cyl, data = mtcars)
)

# Not this
boxplot(mpg ~ cyl, data = mtcars) |> zzplot()
```

This is explicit and unambiguous. The tradeoff is that it breaks the "pipe everything" mental model some users have developed.

Pattern 2: Provide Both APIs

Offer a standard evaluation version alongside the NSE version:

```r
# NSE version - for interactive use
zzplot(plot(x, y))

# SE version - for programmatic use and piping
zzplot_expr(quote(plot(x, y)))

# Or with a string
zzplot_code("plot(x, y)")
```

Pattern 3: Use Quosures (Tidy Eval)

The rlang package provides quosures, which bundle an expression with its environment (Henry and Wickham 2023). This enables more robust expression capture in some contexts:

```
library(rlang)

my_function <- function(expr) {
  quo <- enquo(expr)
  expr_text <- quo_text(quo)
  # ... use the quosure
}
```

However, this still does not solve the pipe problem – `enquo()` captures what it receives, and the pipe has already evaluated the left-hand side.

Pattern 4: Redesign to Avoid NSE

Sometimes the cleanest solution is to not use NSE at all:

```
# Instead of capturing expressions, accept functions
zzplot_fn <- function(plot_fn, ...) {
  png("plot.png")
  plot_fn(...)
  dev.off()
}

# Usage (pipeable with anonymous functions)
\() plot(mtcars$wt, mtcars$mpg) |>
  zzplot_fn()
```

This is less elegant but completely unambiguous about evaluation order.

Checking Our Work: Referential Transparency

This issue reflects a fundamental property of programming languages: *referential transparency*. A function is referentially transparent when any expression can be replaced with its value without changing the program's behavior (Wickham 2019).

```
# Referentially transparent
f <- function(x) x + 1
f(2 + 2)  # Same as f(4)

# NOT referentially transparent
g <- function(x) deparse(substitute(x))
```

```r
g(2 + 2)  # Returns "2 + 2"
g(4)      # Returns "4"
```

NSE functions are not referentially transparent by design: they care about *how* something is expressed, not just *what* value it produces. Pipes assume referential transparency because they pre-compute values.

Hadley Wickham notes in *Advanced R*: "The biggest downside of NSE is that functions that use it are no longer referentially transparent" (Wickham 2019).

Things to Watch Out For

Several situations catch people off guard:

1. **Plot wrappers with logging.** Any function that uses `substitute()` to record what the user typed will capture the wrong thing when piped.
2. **Custom assertion functions.** If an assertion function captures the expression for error messages (like `stopifnot()` does internally), piping changes the error output.
3. **Tidyverse extensions.** Writing functions that wrap `dplyr` verbs with additional `enquo()` logic can behave differently depending on how users call them.
4. **Debugging tools.** Functions that reconstruct the call stack or print "you called f(x)" will show the piped intermediate value rather than the original expression.
5. **Code generation.** Any metaprogramming that builds code from captured expressions will generate different code depending on whether the user piped or nested.

Figure 4: Reflecting on what we have learned.

The distinction between value semantics and expression semantics runs deep in R.

What Did We Learn?

Lessons Learned

Conceptual Understanding:

- The pipe equivalence `f() |> g()` and `g(f())` holds for *value semantics* but fails for *expression semantics*. This is not a bug but a fundamental property of how R evaluates code.
- R's lazy evaluation passes arguments as unevaluated *promises*, which NSE functions can inspect. The pipe forces evaluation before the handoff, destroying the promise.
- Referential transparency is the key concept: pipes assume it, and NSE functions violate it by design.
- The tension between eager evaluation (pipes), lazy evaluation (promises), and non-standard evaluation (expression inspection) is inherent in R's design.

Technical Skills:

- Tracing the evaluation order difference between `capture_expr(sqrt(16))` and `sqrt(16) |> capture_expr()` step by step clarifies the underlying mechanism.
- Distinguishing between functions that are pipe-safe (value-based) and those that are not (expression-based) requires checking whether they use `substitute()`, `match.call()`, or `enquo()`.
- Four design patterns exist for building pipe-compatible APIs: accepting the limitation, dual APIs, quosures, and NSE avoidance.
- Tidyverse functions work with pipes despite using NSE internally because they evaluate captured expressions in data contexts rather than inspecting them for the calling expression.

Gotchas and Pitfalls:

- The native pipe (`|>`) and magrittr pipe (`%>%`) both have this limitation. Switching pipe operators does not solve the problem.
- The failure is *silent*: the piped version does not throw an error; it simply captures the wrong thing. This makes the bug difficult to detect without explicit testing.
- Documentation for NSE functions rarely mentions pipe incompatibility. Authors of such functions should document the limitation explicitly.
- Quosures (`enquo()`) do not solve the pipe problem – they still receive the pre-evaluated value when the call is piped.

Limitations

- This post focuses on expression capture via `substitute()` and related tools. Other forms of NSE (such as formula evaluation or `eval(parse())`) may interact with pipes differently and are not covered.
- The examples use simple, isolated functions. In production code, the interaction between pipes, NSE, and environments can be more complex, particularly with nested function calls.
- This post did not benchmark the performance implications of the different workaround patterns. For most use cases the performance difference is negligible, but this is not verified.

- The discussion treats `|>` and `%>%` as equivalent for this problem. There are other differences between them (placeholder syntax, anonymous function support) that are outside the scope of this post.
- This analysis is based on R 4.1+ behavior. Earlier versions of R did not have the native pipe, and the magrittr pipe has evolved over time.

Opportunities for Improvement

1. Build a comprehensive test suite that validates pipe-safety for a library of utility functions, automatically flagging any function that uses NSE internally.
2. Investigate whether R's native pipe could be extended to support a "lazy" mode that preserves promise semantics, and assess the trade-offs.
3. Create a static analysis tool (perhaps using the `codetools` package) that warns when NSE functions appear downstream of a pipe operator.
4. Explore how other languages with pipe operators (Elixir, F#, Julia) handle the tension between eager evaluation and expression capture.
5. Document this limitation in the context of specific tidyverse extension patterns, providing a cookbook of pipe-safe alternatives for common NSE use cases.

Wrapping Up

The pipe equivalence `f() |> g()` and `g(f())` holds for value semantics but fails for expression semantics. This is not a bug in R or the pipe operators; it is a fundamental tension between:

- **Eager evaluation** (pipes compute left-to-right)
- **Lazy evaluation** (functions receive unevaluated promises)
- **Non-standard evaluation** (functions inspect the expression, not just the value)

The most valuable aspect of this investigation is the understanding it provides of R's evaluation model. The pipe is not "just sugar"; it changes the evaluation semantics in a way that matters for a specific but important class of functions.

For anyone writing functions that use `substitute()`, `match.call()`, or similar metaprogramming tools: documenting that piping will not work as expected is warranted. For anyone using such functions: the distinction between wrapping `f(g(x))` and piping `x |> g() |> f()` is a semantic difference, even when both produce the same final value.

In conclusion, four points merit emphasis. First, pipes force eager evaluation, destroying the lazy promises that NSE functions depend on. Second, the failure is silent: no error is raised, but the output is wrong. Third, four design patterns exist for working around this limitation: accepting it, providing dual APIs, using quosures, or avoiding NSE entirely. Fourth, understanding this distinction is a prerequisite for effective R metaprogramming.

See Also

Related Posts

- Unix Command-Line Workspace Setup for Data Science Development

: Terminal and Zsh configuration for R workflows

Key Resources

- Non-standard evaluation in *Advanced R* (1st ed.) by Hadley Wickham
- Evaluation chapter in *Advanced R* (2nd ed.)
- Tidy Evaluation book by the tidyverse team
- Standard and Non-Standard Evaluation in R by Brodie Gaslam
- Differences between the base R and magrittr pipes on the tidyverse blog
- Design tradeoffs in magrittr
- Understanding Non-Standard Evaluation, Part 1 by Thomas Adventureson

References

Bache, Stefan Milton, and Hadley Wickham. 2022. *Magrittr: A Forward-Pipe Operator for R*. https://CRAN.R-project.org/package=magrittr.

Henry, Lionel, and Hadley Wickham. 2023. *Rlang: Functions for Base Types and Core R and Tidyverse Features*. https://CRAN.R-project.org/package=rlang.

Wickham, Hadley. 2019. *Advanced R*. 2nd ed. Chapman; Hall/CRC. https://adv-r.hadley.nz/.

Wickham, Hadley, and Lionel Henry. 2023. *Differences Between the Base R and Magrittr Pipes*. Tidyverse blog. https://tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/.

Reproducibility

This post is primarily conceptual with minimal executable code. The few code chunks that do execute require only base R (version 4.1 or later for native pipe support).

```
sessionInfo()
```

```
R version 4.6.1 (2026-06-24)
Platform: aarch64-apple-darwin25.4.0
Running under: macOS Tahoe 26.6.2

Matrix products: default
BLAS:   /opt/homebrew/Cellar/openblas/0.3.34/lib/libopenblasp-r0.3.34.dylib
```

```
LAPACK: /opt/homebrew/Cellar/r/4.6.1/lib/R/lib/libRlapack.dylib;  LAPACK version 3.12.1

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.6.1  fastmap_1.2.0   cli_3.6.6       tools_4.6.1
 [5] htmltools_0.5.9 parallel_4.6.1  otel_0.2.0      yaml_2.3.12
 [9] rmarkdown_2.31  knitr_1.51      jsonlite_2.0.0  xfun_0.58
[13] digest_0.6.39   rlang_1.3.0     evaluate_1.0.5
```

Files in this analysis:

```
analysis/report/index.qmd    This blog post
references.bib                Bibliography (BibTeX)
```

Let's Connect

Have questions, suggestions, or spot an error? Let me know.

- **GitHub:** rgt47
- **Twitter/X:** (**rgt47?**)
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** Contact form

I would enjoy hearing from you if:

- An error or a better approach to any of the code in this post comes to mind.
- There are suggestions for topics to see covered.
- The interest is in discussing R programming, data science, or reproducible research.
- There are questions about anything in this tutorial.
- The goal is simply to say hello and connect.

Related posts in this cluster

This post is part of the *R Language and Metaprogramming* series. Recommended reading order:

1. **Post 62: The Pipe Equivalence Myth** (this post)
2. Post 63: Dynamic Column Names: Seven Approaches Compared