Tidy evaluation (hygienic fexprs)

Lionel Henry and Hadley Wickham | RStudio

Tidy evaluation

- Result of our quest to harness fexprs (NSE functions)
  - Based on our experience with base R fexprs
  - tidyeval takes this experience + solves **hygiene** problems
- fexpr = function with **pass-by-expression** semantics

- Model formulas
- `base::subset()` and `transform()`
- dplyr, ggplot2

fexprs versus macros

Similar to macros (unevaluated arguments) but different

fexprs	macros
Run-time	Compile-time
Return a value	Code expansion
First-class	Transient
Not compilable	Compilable

Kent M. Pitman, "Special Forms in Lisp", *Proceedings of the 1980 ACM Conference on Lisp and Functional Programming*, 1980
Mitchell Wand, "The Theory of Fexprs is Trivial", *Lisp and Symbolic Computation*, 10(3), 1998
John N. Schutt, *Fexprs as the basis of Lisp function application*, Worcester Polytechnic Institute, 2010

fexprs versus macros

- **fexprs** were abandoned in the 1980s
  - Hard to compile (for same reason: `quote()` + `eval()` is evil)
  - Weird semantics (dynamic scope and no first-class envs)
- **macros** benefit from more than 50 years of research
  - Hygiene is a big topic
  - We'll see it's important for fexprs as well
- But fexprs lived on in New S and R!
  - What did we learn?

What does base R teach us about fexprs?

- **Overscoping:** evaluate expressions in data context
- **Formulas:** systematic capture of environment

Overscoping

- Code is delayed to be evaluated in **data context**
- **Original context** is still kept in scope
- **Evaluation** makes sure we still have full R semantics

⟶ Major idiom that gives R its identity

Overscoping

- Code is delayed to be evaluated in **data context**
- **Original context** is still kept in scope
- **Evaluation** makes sure we still have full R semantics

⟶ Major idiom that gives R its identity

Model formulas

```r
var <- 1:32
lm(disp ~ var + as.factor(cyl), mtcars)
```

Overscoping

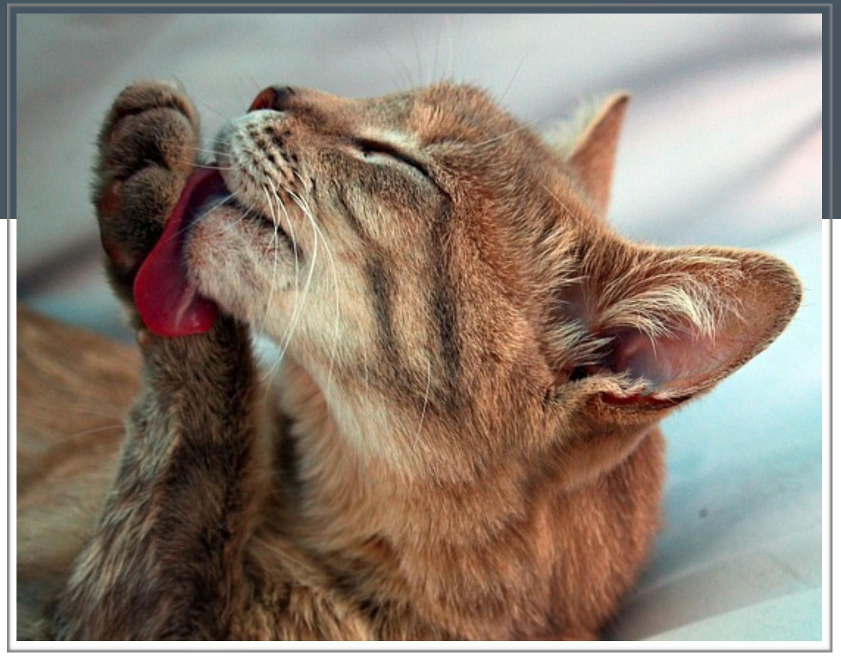
- Code is delayed to be evaluated in **data context**
- **Original context** is still kept in scope
- **Evaluation** makes sure we still have full R semantics

⟶ Major idiom that gives R its identity

Datawise operations

```
var <- 6
subset(mtcars, cyl == var)
with(mtcars, cyl + var)
```

Hygiene



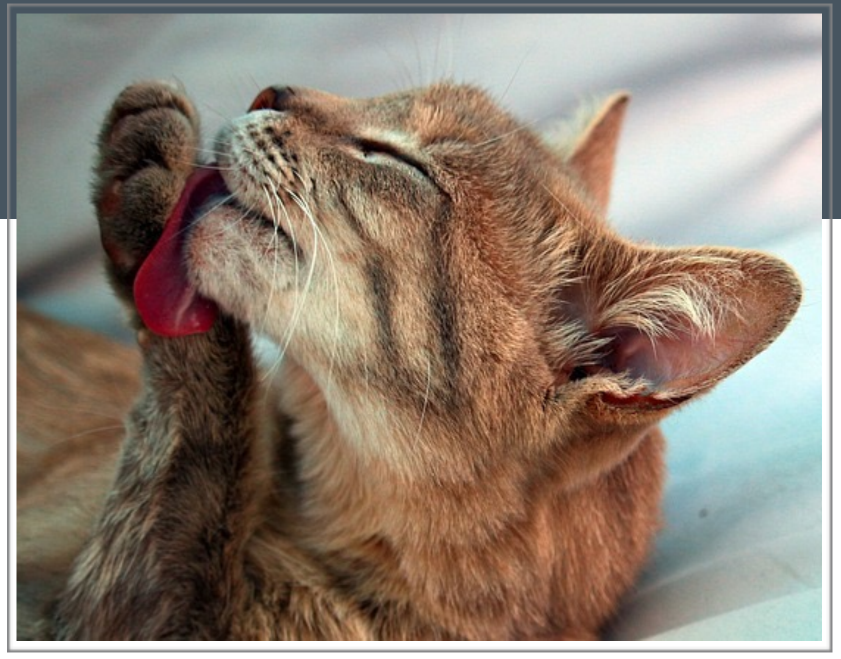
Keeping the context around $\longrightarrow$ notion of **hygiene**

Symbols should be looked up in the context where they appear

Hygiene fosters **locality of reasoning**

```
var <- 6
subset(mtcars, cyl == var)
with(mtcars, cyl + var)
```

Hygiene



- Macro expansion can hide local variables
- For fexprs hygiene is about expansion *and* evaluation
- In R hygiene is complicated by overscoping
  $\longrightarrow$ a proper overscope is crucial for consistent semantics

- **data**
- **context**

```
var <- 6
subset(mtcars, cyl == var)
with(mtcars, cyl + var)
```

Overscoping

Making an overscope

- Turn data to environment
- Set original context as parent

Hence `eval()` takes *`envir`* and *`enclos`* arguments

```
eval(expr, data, environment())
```

We need the original environment!

⟶ **formulas** for explicit capture;
easy and safe to pass around

⟶ **`parent.frame()`** for substituted capture

substitute()

Implicit capture

```
quote <- function(x) {
  substitute(x)
}

quotes <- function(...) {
  eval(substitute(alist(...)))
}
```

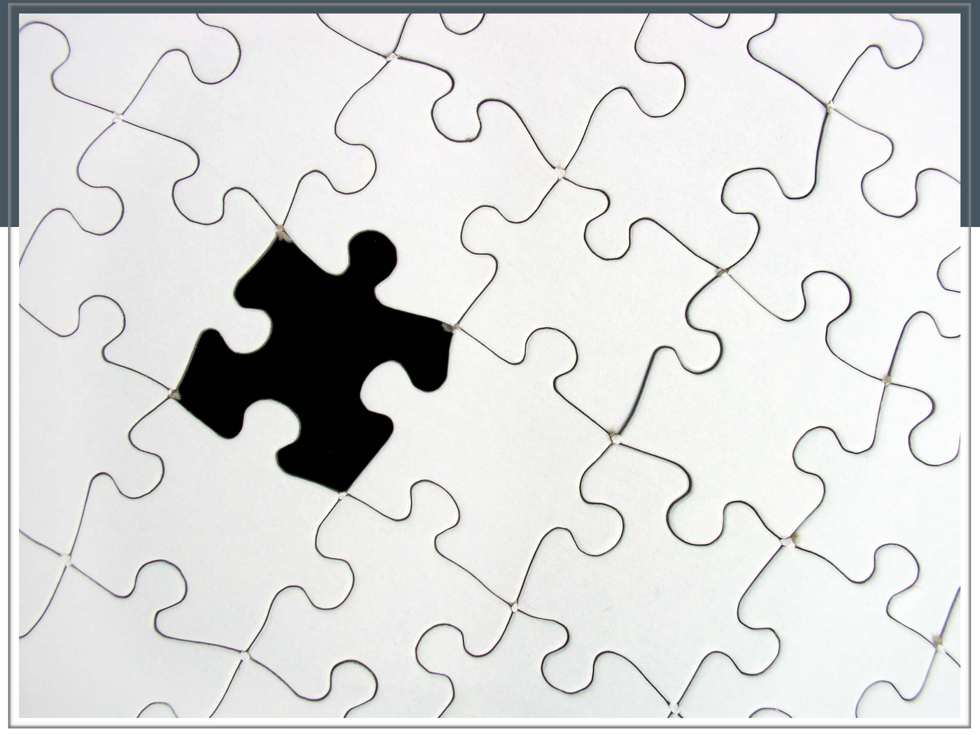
Code expansion

```
listify <- function(x, y) {
  substitute(list(x, y))
}

listify(foo, bar())
#> list(foo, bar())
```

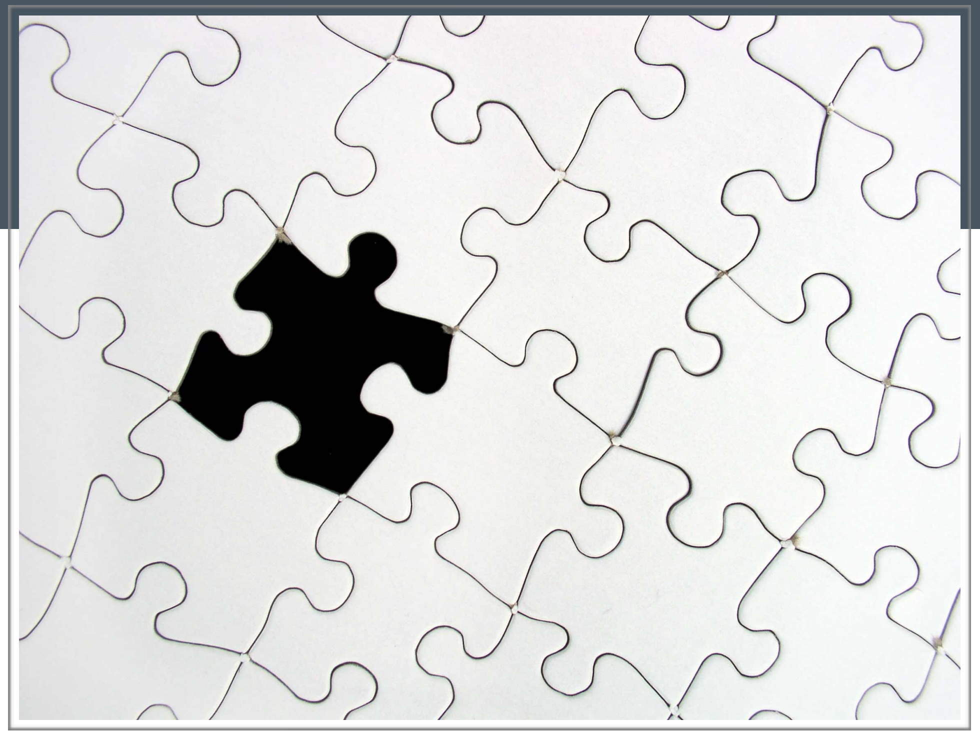
- Returns a bare expression
- Has to be paired with `parent.frame()`

What's missing?



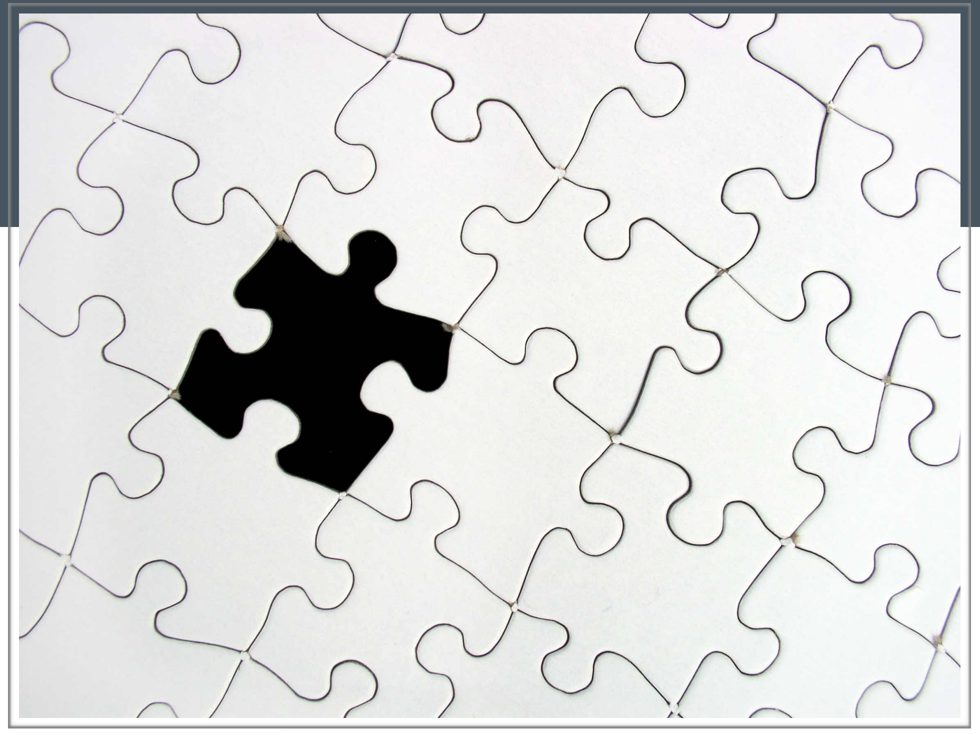
- Systematic capture of context
- Hygienic code expansion
- Opting in and out the overscope

What's missing?



- Systematic capture of context
- Hygienic code expansion
- Opting in and out the overscope

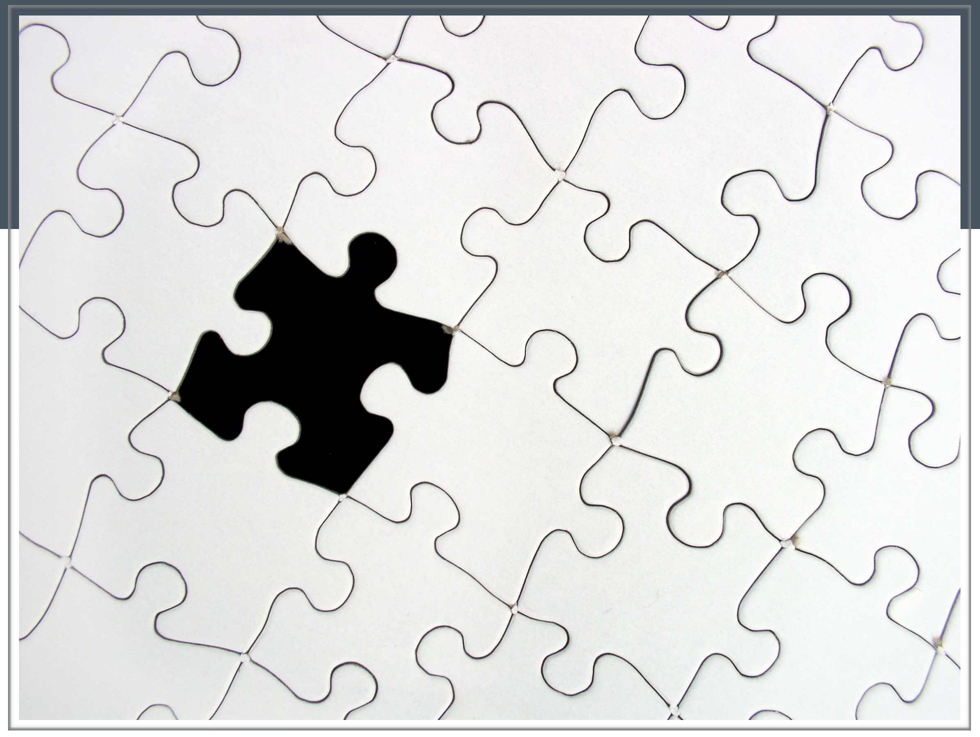
`substitute()`



Is `parent.frame()` always the hygienic context?

- What if arguments are **forwarded**?
- What if expanded code refers to **local symbols**?

substitute()

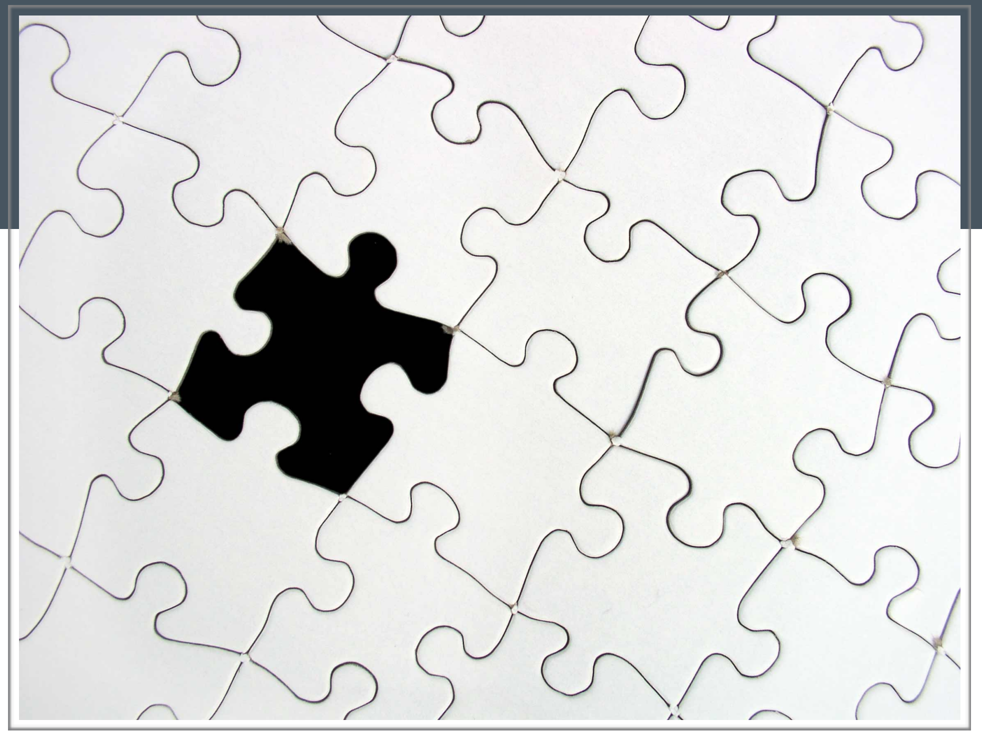


What if arguments are **forwarded**

```
transform <- function(data, ...) {
  expr <- substitute(list(...))
  vals <- eval(expr, data, parent.frame())
  *truncated*
}

wrapper <- function(data, ...) {
  var <- "wrong"
  transform(data, ...)
}
```

substitute()



What if arguments are **forwarded**

```
transform <- function(data, ...) {
  expr <- substitute(list(...))
  vals <- eval(expr, data, parent.frame())
  # *truncated*
}

wrapper <- function(data, ...) {
  var <- "wrong"
  transform(data, ...)
}
```

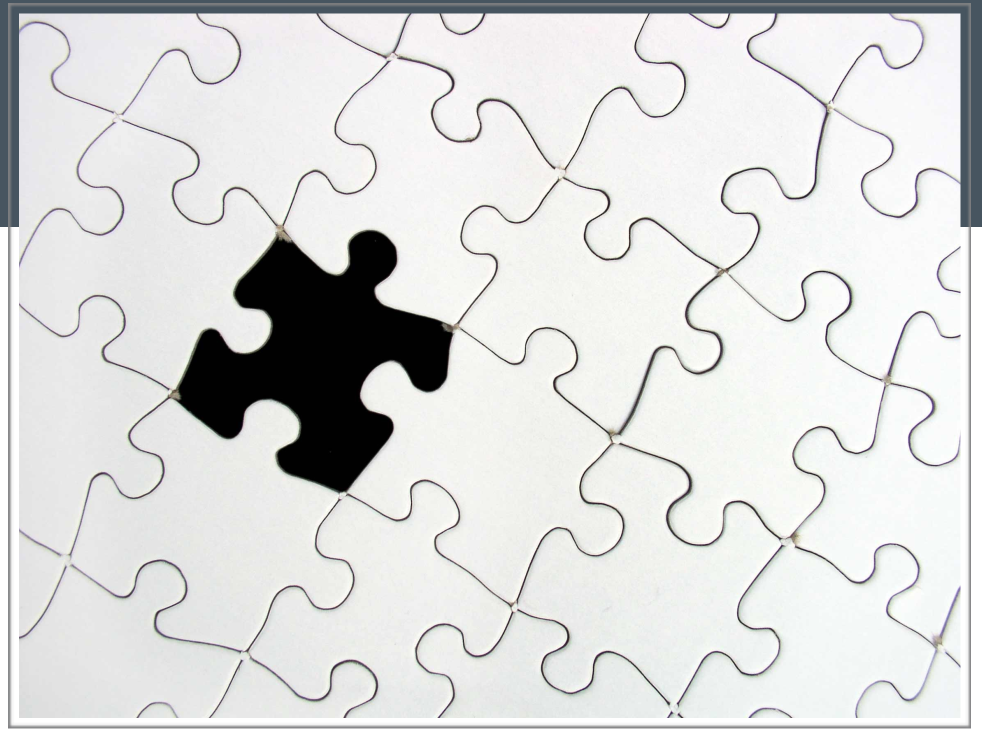
```
var <- 10

transform(mtcars, new = cyl * var)

wrapper(mtcars, new = cyl * var)

local({
  var <- 1000
  dfs <- list(mtcars, mtcars)
  lapply(dfs, transform, new = cyl * var)
})
```

substitute()



What if expanded code refers to **local symbols**?

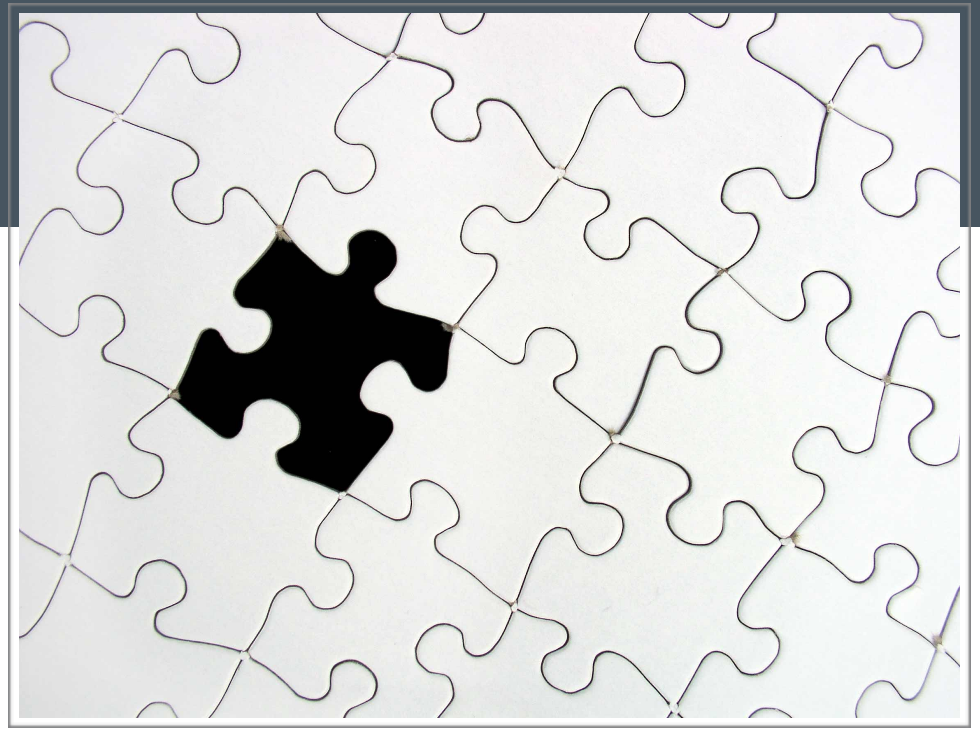
```
ll <- base::list

transform <- function(data, ...) {
  expr <- substitute(ll(...))
  vals <- eval(expr, data, parent.frame())
  *truncated*
}
```

This issue is compounded by forwarded arguments

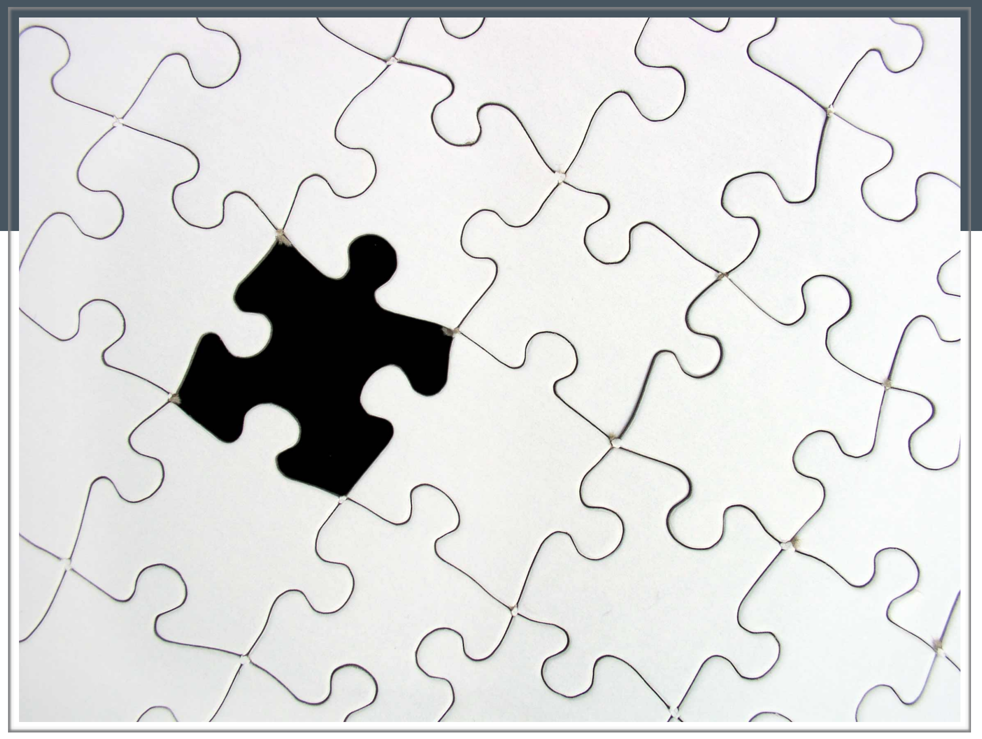
⟶ Lack of **hygienic code expansion**

What's missing?



- Systematic capture of context
- Hygienic code expansion
- Opting in and out the overscope

substitute()



How to **opt out** of the overscope?

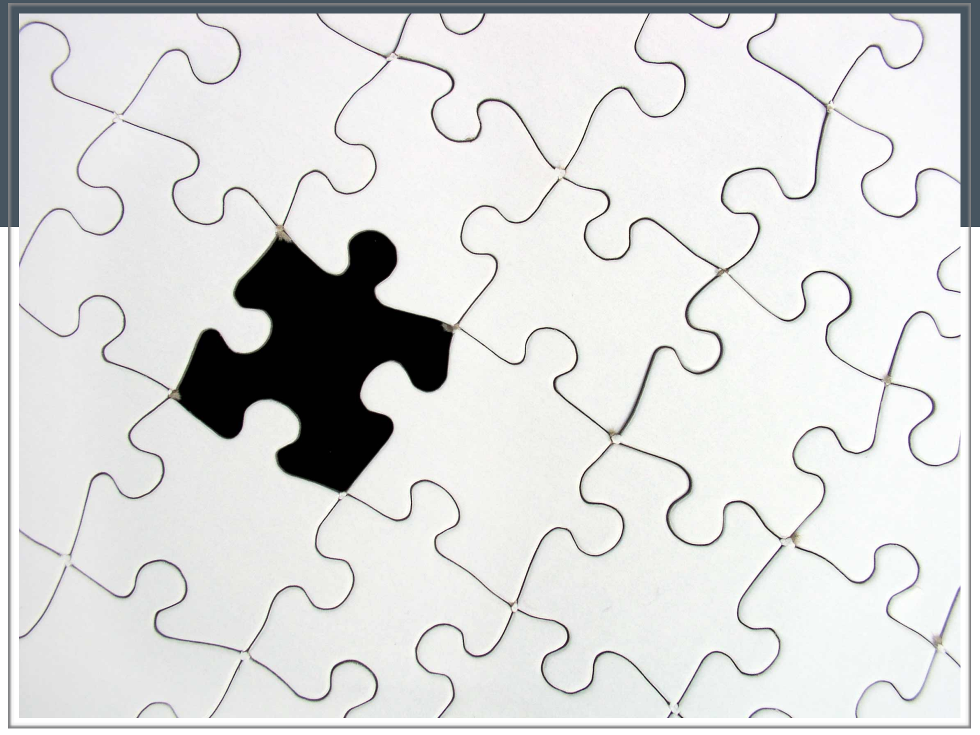
```
var <- 10
mtcars$var <- seq_len(nrow(data))

transform(mtcars, new = cyl * var)
```

The overscope is a *moving part*

- For data analysis, no worries
- For functions, need a bit more hygiene

substitute()



How to **opt in** the overscope?

⟶ Parameterisation of fexprs against overscope

```
var <- as.name("disp")
transform(mtcars, new = cyl * var)
#> Error in cyl * var :
#> non-numeric argument to binary operator
```

Why program against the quoted expression?

- No context-switch when extracting function from script
- Performance and semantics when fexpr is an interface

Tidy evaluation

- Systematic capture of context
- Hygienic code expansion
- Opting in and out the overscope

Quosures

Quasiquotation

Quosures

Just like formulas, quosures

- bundle
  - a quoted expression
  - a lexical enclosure
- are first-class (easy to pass down to other functions, …)

But they are not literals!

- Like symbols and function calls they **represent a value**
- Evaluate in their **own environments** (possibly overscoped)
- They have semantics of **reified promises**

Quosures

```r
quosure <- local({
  var <- "foo"
  quo(toupper(var))
})


eval(quosure)
#> <quosure: local>
#> ~toupper(var)


var <- "other"
eval_tidy(quosure)
#> [1] "FOO"
```

quo( ) creates a local quosure

Subclass of formula that self-quotes under evaluation…

… but self-evaluates under *tidy* evaluation

Quosures

```
fexpr <- function(x) enquo(x)
fexpr(foo)
#> <quosure: global>
#> ~foo

variadic <- function(...) quos(...)
variadic(foo, bar)
#> [[1]]
#> <quosure: global>
#> ~foo

#> [[2]]
#> <quosure: global>
#> ~bar
```

- `enquo()` turns argument to quosure
- `quos()` turns forwarded arguments to quosures

Quasiquotation

- Useful for code expansion (e.g. lisp macroexp)
- We enable it in *all* fexprs $\longrightarrow$ tamable overscope

```
var <- "foo"
quo(list(UQ(var)))
#> <quosure: global>
#> ~list("foo")

quo(list(UQS(letters[1:3])))
#> <quosure: global>
#> ~list("a", "b", "c")
```

- `UQ()` to unquote and inline
- `UQS()` to unquote and splice
- `!!` and `!!!` syntax

Hygienic code expansion

```r
var <- "foo"

inner <- local({
  var <- "bar"
  quo(var)
})
nested <- local({
  concat <- c
  quo(concat(var, UQ(inner)))
})
```

```r
nested
#> <quo>
#> ~concat(var, ~var)

eval_tidy(nested)
#> [1] "foo" "bar"
```

⟶ Full lexical scope within expanded expression!

Quosure overscoping

Quosures evaluated within a given expression can be **overscoped**

```
nested
#> <quosure: local>
#> ~concat(var, ~var)

data <- list(var = "boo!")

eval_tidy(nested, data)
#> [1] "boo!" "boo!"
```

We'll soon introduce *safe quosures*

- Never evaluated within overscope
- Laziness + safety

Taming the overscope

Let's use `dplyr::mutate()` instead of `transform()`

Opting out of the overscope

```
cyl <- 10
mutate(mtcars, new = cyl * (!! cyl))
```

Opting in

```
var <- as.name("disp")
mutate(mtcars, new = cyl * (!! var))
mutate(mtcars, new = cyl * disp)
```

Opting in and out

$\downarrow$

Hygienic overscoping

Summary

To sum things up, let's fix `transform()`

- Capture dots in quosures
- Hygienic expansion with unquote-splice
- Quosure-friendly evaluation

```
transform <- function(data, ...) {
  expr <- quo(list(UQS(quos(...))))
  vals <- eval_tidy(expr, data)
  # truncated
}
```

- Tidy capture
- Tidy evaluation
- Tidy overscope

(where tidy means hygienic)