V12 - Meta-Programmierung

31. Mai 2021

Contents


```
library(tidyverse)
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.2     v dplyr   1.0.6
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
library(rlang)
##
## Attaching package: 'rlang'
## The following objects are masked from 'package:purrr':
##
##     %@%, as_function, flatten, flatten_chr, flatten_dbl, flatten_int,
##     flatten_lgl, flatten_raw, invoke, list_along, modify, prepend,
##     splice
library(lobstr)
```

1 Intro

Einige Funktionen haben ein sonderbares Verhalten:

Wie kann `library()` ohne " funktionieren?

```
rlang # keine Variable
## Error in eval(expr, envir, enclos): object 'rlang' not found
library(rlang) # funktioniert (ohne "")
```

Woher kennt `tibble()` den Variablennamen der Argumente?

```
numbers <- 1:26
tibble(numbers, letters)
## # A tibble: 26 x 2
##    numbers letters
##      <int> <chr>
##  1       1 a
##  2       2 b
##  3       3 c
##  4       4 d
##  5       5 e
##  6       6 f
##  7       7 g
##  8       8 h
##  9       9 i
## 10      10 j
## # ... with 16 more rows
```

Wie können `dplyr`-Funktionen Spalten wie Variablen verwenden?

```
tb <- tibble(x = 1:5, y = 5:1)
filter(tb, x > y)
## # A tibble: 2 x 2
##       x     y
##   <int> <int>
## 1     4     2
## 2     5     1
```

Antwort: **Meta-Programmierung**!

2 Ausdrücke

In der Meta-Programmierung wird R-Code selbst zu einem Objekt, das wir manipulieren und auswerten können.

Ein **Ausdruck** ist ein syntaktisch korrekter R-Funktionsaufruf, zB `3+5*4`, `x <- mean(1:100)`, `f(g(h(x),y),z)`. Dies ist unabhängig von Namensbindungen.

Um einen Ausdruck in einem R-Objekt zu speichern, verwenden wir `rlang::expr()`. `rlang::expr()` verhindert die sofortige Auswertung ihres Argumentes. Dieser Vorgang wird **Defusing** genannt.

```
x <- expr(3+5*4)
x
## 3 + 5 * 4
y <- expr(x <- mean(1:100))
y
## x <- mean(1:100)
```

```r
z <- expr(f(g(h(x),y),z))
z
## f(g(h(x), y), z)
```

```r
expr(f(x-)) # ERROR: nicht syntaktisch korrekt
```

Mit `rlang::exprs()` werden mehrere Ausdrücke in eine Liste von Ausdrücken aufgenommen. `c()` fügt mehrere Ausdrücke zu einer Liste zusammen.

```r
str(exprs(1 + 2, f(x))) # zwei Ausdrücke
## List of 2
##  $ : language 1 + 2
##  $ : language f(x)
expr({1 + 2; f(x)}) # ein Ausdruck
## {
##     1 + 2
##     f(x)
## }
str(exprs(name1 = 1 + 2, name2 = f(x))) # auch als benannte Liste
## List of 2
##  $ name1: language 1 + 2
##  $ name2: language f(x)
c(expr(1 + 2), expr(f(x)))
## [[1]]
## 1 + 2
##
## [[2]]
## f(x)
```

`eval()` wertet Ausdruck-Objekte aus.

```r
x <- expr(3+5*4)
y <- expr(x <- mean(1:100))
z <- expr(f(g(h(x),y),z))
eval(x)
## [1] 23
eval(y)
x
## [1] 50.5
eval(z)
## Error in f(g(h(x), y), z): could not find function "f"
```

2.1 Bestandteile von Ausdrücken

Ausdrücke setzen sich zusammen aus **Funktionsaufrufen**, **Konstanten** und **Symbolen**.

Konstanten sind Werte, für die keinerlei Namensauflösung vollzogen werden muss, zB `NULL`, `NA`, `FALSE`, `42L`, `5.5`, `"asdf"`, nicht jedoch `F` oder `pi`.

Für Konstanten ist `rlang::expr()` die Identität. Es wird also nicht unterschieden zwischen dem Ausdruck `1` und dem Wert `1`.

```r
typeof(expr(1))
## [1] "double"
c(identical(FALSE, expr(FALSE)), identical(42L, expr(42L)), identical(F, expr(F)))
## [1]  TRUE  TRUE FALSE
```

Der Begriff **Symbol** bezeichnet einen Variablennamen. Da beim Erstellen eines Ausdrucks nur auf syntaktische Korrektheit geprüft wird, nicht jedoch auf Auswertbarkeit, können Symbole auch Namen sein, zu denen kein Wert gebunden ist.

Symbole haben Typ `symbol` und Klasse `name`.

```
x <- expr(a_variable_name)
c(typeof(x), class(x))
## [1] "symbol" "name"
```

Durch **Funktionsaufrufe** können Symbole, Konstanten und andere Funktionsaufrufe zu komplexeren Ausdrücken zusammengestellt werden.

Funktionsaufrufe haben Typ `language` und Klasse `call`.

```
a <- expr(f(g(h(x),y),z))
c(typeof(a), class(a))
## [1] "language" "call"
b <- expr(1 + 2)
c(typeof(b), class(b))
## [1] "language" "call"
```

2.2 Syntaxbaum

Jedem Ausdruck liegt eine Baumstruktur zugrunde, die angibt, welche Funktion mit welchen Argumenten ausgeführt werden soll. Sie wird als **Syntaxbaum** bezeichnet.

Der Syntaxbaum eines Ausdrucks ist ein gewurzelter Baum.

Die Wurzel ist die äußerste Funktion. In `f(g(x, h(), 1), x)` ist die Wurzel `f()`.

Die Kind-Knoten eines Knotens stehen für die Argumente der Funktion. Die Kinder des Knotens mit der Funktion `f()` sind `g(x, h(), 1)` und `x`.

Die Blätter eines Syntaxbaumes sind Symbole, Konstanten oder Funktionen ohne Argumente. Im Ausdruck `f(g(x, h(), 1), x)` sind die Blätter `x`, `h()`, `1` und `x`.

Achtung: Die Reihenfolge der Kinder spielt eine Rolle (`f(x,y)` $\neq$ `f(y,x)`).

Die Funktion `lobstr::ast()` zeigt den Syntaxbaum eines Ausdrucks auf der Konsole an.

```
ast(f(g(x, h(), 1), x))
## o-f
## +-o-g
## | +-x
## | +-o-h
## | \-1
## \-x
```

`language`-Objekte sind im Wesentlichen verschachtelte Listen:

- Umwandlung zwischen `language` und `list` mit `as.call()` und `as.list()`
- Zugriff auf Elemente mit `[`, `[<-`, `[[`, `[[<-`
- Element an Position 1 ist Funktionsname, weitere Elemente sind Argumente

```
x <- expr(f(3+5,4))
x
## f(3 + 5, 4)
str(as.list(x))
## List of 3
##  $ : symbol f
```

```
##  $ : language 3 + 5
##  $ : num 4
x[[1]] <- expr(g)
x
## g(3 + 5, 4)
x[c(1,3)]
## g(4)

a <- expr(f(g(x, h(), 1), x))
a[[1]]
## f
a[[2]]
## g(x, h(), 1)
a[[3]]
## x
a[[2]][[1]]
## g
a[[2]][[2]]
## x
a[[2]][[3]]
## h()
a[[2]][[4]]
## [1] 1
a[[2]][[3]][[1]]
## h
typeof(a[[2]][[3]]) # h()
## [1] "language"
typeof(a[[2]][[3]][[1]]) # h
## [1] "symbol"
# siehe auch View(a) in RStudio
```

Hier können wir Indexvektoren für `[[` sinnvoll einsetzen. Erinnerung `x[[c(1,2,3)]]` entspricht `x[[1]][[2]][[3]]`.

```
a[[c(2, 3, 1)]]
## h
```

2.3 Ausdrücke und Strings

Um einen Ausdruck in seine Repräsentation als String zu wandeln, nutze `rlang::expr_text()`.

```
z <- expr(f(g(h(x),y),z))
z_str <- expr_text(z)
typeof(z_str)
## [1] "character"
z_str
## [1] "f(g(h(x), y), z)"
```

Um einen String, der syntaktisch korrekten R-Code enthält, in einen Ausdruck-Objekt zu konvertieren, nutze `rlang::parse_expr()`.

Enthält der String mehrere Befehle (durch `;` oder Zeilenumbruch getrennt) kann mit der Funktion `rlang::parse_exprs()` eine Liste von Ausdrücken erzeugt werden.

```
y <- parse_expr("x <- 1+2")
typeof(y)
```

```
## [1] "language"
eval(y)
x
## [1] 3
z <- parse_exprs("x <- 4\nprint(x+5)")
str(z)
## List of 2
##  $ : language x <- 4
##  $ : language print(x + 5)
```

2.4 Base-R

In Base-R gibt es einen eigenen Datentyp für Listen von Ausdrücken. Dieser hat verwirrenderweise den Namen `"expression"`.

`parse()` ist die Base-R-Variante von `rlang::parse_exprs()`.

```
typeof(parse(text="1+2;3-4"))
## [1] "expression"
```

`rlang` verwendet den Datentyp `expression` nicht. Stattdessen wird eine Liste (`list`) von Ausdrücken (`language`, `symbol` oder Konstante) verwendet.

```
# in rlang
b <- parse_exprs("1+2;3;a")
typeof(b)
## [1] "list"
str(b)
## List of 3
##  $ : language 1 + 2
##  $ : num 3
##  $ : symbol a
b[[1]]
## 1 + 2

# in Base-R:
a <- parse(text="1+2;3;a")
typeof(a)
## [1] "expression"
str(a)
##   expression(1 + 2, 3, a)
a[[1]]
## 1 + 2
typeof(a[[1]])
## [1] "language"

# Vergleich:
identical(a, b)
## [1] FALSE
mapply(identical, a, b)
## [1] TRUE TRUE TRUE
```

3 Ausrdrücke verwenden

3.1 Argumente

Wir möchten nun aufnehmen, mit welchem Ausdruck ein Nutzer einer Funktion ein Argument übergibt, zB beim Aufruf von `f(1+2)` den Ausdruck `1+2` anstatt den Wert `3`. Mit den bisher gelernten Funktionen schaffen wir es nicht.

```
f <- function(x) expr(x)
f(1 + 2)
## x
```

`rlang::enexpr()` gibt den Ausdruck des übergebenen Arguments zurück.

```
h <- function(x) enexpr(x)
h(1 + 2)
## 1 + 2
```

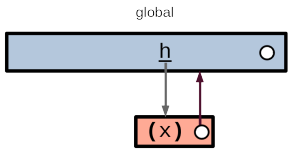
Erinnerung: Im Inneren einer Funktion sind die Argumente Versprechen, solange sie noch nicht ausgewertet worden sind. Ein Versprechen besteht aus einem Ausdruck und einer Umgebung, in der der Ausdruck ausgewertet werden soll. Mit `rlang::enexpr()` erhalten wir den Ausdruck ohne das Versprechen einzulösen.

```
h <- function(x) {
  env_print(current_env())  # <lazy> zeigt Versprechen an
  res <- enexpr(x)
  env_print(current_env())
  force(x)
  env_print(current_env())
  res
}
y <- h(1 + 2)
## <environment: 0000000020C3F050>
## parent: <environment: global>
## bindings:
##  * x: <lazy>
## <environment: 0000000020C3F050>
## parent: <environment: global>
## bindings:
##  * res: <language>
##  * x: <lazy>
## <environment: 0000000020C3F050>
## parent: <environment: global>
## bindings:
##  * res: <language>
##  * x: <dbl>
y
## 1 + 2
```

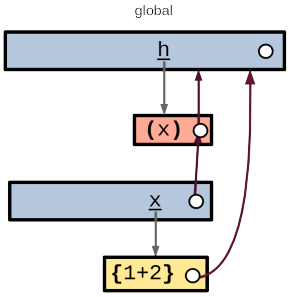
Der gleiche Code mit Umgebungsdiagrammen:

```
h <- function(x) {
  # 2
  res <- enexpr(x)
  # 3
  force(x)
  # 4
  res
}
# 1
y <- h(1 + 2)
```

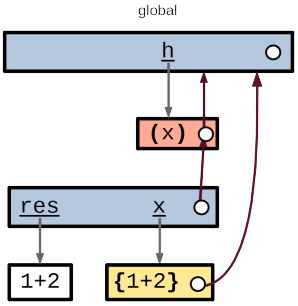
```
# 5
y
## 1 + 2
```



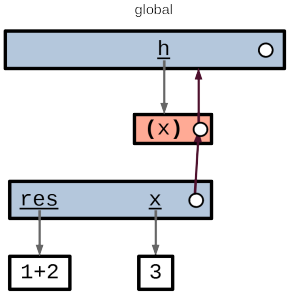
1



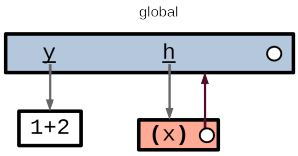
2



3



4



5

Wir müssen `rlang::enexpr()` aufrufen, bevor das Versprechen eingelöst wird. `rlang::enexpr()` selbst löst das Versprechen nicht ein.

```
h <- function(x) {
  print(enexpr(x))
  print(enexpr(x))
  force(x)
  print(enexpr(x))
}
h(1+2)
## 1 + 2
## 1 + 2
## [1] 3
```

Um die Elemente des `...`-Arguments als Liste von Ausdrücken aufzunehmen, kann `rlang::enexprs()` genutzt werden.

```
f <- function(...) enexprs(...)
str(f(x="asdf", a, z=5+3))
## List of 3
##  $ x: chr "asdf"
##  $  : symbol a
##  $ z: language 5 + 3
```

Als Anwendungsbeispiel geben wir die Ausdrücke der Argument als Text aus.

```
direct_paste <- function(...) {
  dots <- enexprs(...)
  str_c(sapply(dots, expr_text), collapse = " ")
}
```

```r
direct_paste(direkte, Ausgabe)
## [1] "direkte Ausgabe"
```

3.2 Auswertung und Umgebungen

Die Auswertung eines Ausdrucks mit `eval()` geschieht in der aktuellen Umgebung, außer wir übergeben als zweites Argument eine andere Umgebung.

```r
x <- 5
x_expr <- expr(x * 2)
eval(x_expr)
## [1] 10
eval(x_expr, env(x = 4))
## [1] 8

y <- 3
eval(
  expr(x + y),
  env(`+` = `-`)
)
## [1] 2
```

3.3 Quosures

Da zu der Auswertung eines Ausdrucks also immer auch eine Umgebung gehört, in der der Ausdruck ausgeführt wird, stellt das Paket `rlang` die Datenstruktur *Quosure* bereit, die beides zusammenhält.

Ein *Quosure* ist ein S3-Objekt basierend auf einem Ausdruck-Objekt mit einem Attribut `.Environment`, welches die Umgebung speichert.

`quo()` und `quos()` sind analog zu `expr()` und `exprs()` und nehmen die aktuelle Umgebung mit auf.

```r
q <- quo(1+x)
q
## <quosure>
## expr: ^1 + x
## env:  global
typeof(q)
## [1] "language"
attributes(q)
## $class
## [1] "quosure" "formula"
##
## $.Environment
## <environment: R_GlobalEnv>
```

Mit `quo_get_expr()` und `quo_get_env()` wird auf den Ausdruck bzw die Umgebung eines Quosures zugegriffen.

`eval_tidy(q)` für eine Quosure `q` entspricht `eval(quo_get_expr(q), quo_get_env(q))`.

```r
f <- function() {
  x <- 10
  quo(1+x)
}
x <- 100
q <- f()
```

```
eval(quo_get_expr(q))
## [1] 101
eval_tidy(q)
## [1] 11
```

`enquo()` und `enquos()` sind analog zu `enexpr()` und `enexprs()` und nehmen die Umgebung auf, in der die Argumente ausgewertet werden. Sie wandeln also ein Versprechen (mit Ausdruck und Umgebung) in ein Quosure.

```
foo <- function(x) {
  y <- 1
  x <- enexpr(x)
  eval(x)
}
z <- 10
foo(z * 2)
## [1] 20
y <- 100
foo(y * 2)
## [1] 2

foo2 <- function(x) {
  y <- 1
  x <- enexpr(x)
  eval(x, caller_env())
}
y <- 100
foo2(y * 2)
## [1] 200

foo3 <- function(x) {
  y <- 1
  x <- enquo(x)
  eval_tidy(x)
}
y <- 100
foo3(y * 2)
## [1] 200
```

`eval_tidy()` erlaubt es sogenannte **Data-Masks** zu verwenden: Wird `eval_tidy()` als zweites Argument eine Liste benannter Elemente übergeben, wird bei der Auswertung des Ausdrucks zuerst dort nach Namen gesucht und erst danach in der Umgebung des Quosures.

```
x <- 1
y <- 10
eval_tidy(quo(x+y), list(y=100))
## [1] 101
```

Auf dieser Mechanik basieren viele Funktionen im Tidyverse, bei denen sich Spalten in Tibbles wie Variablen verwenden lassen.

```
tb <- tibble(x=1:3, y=11:13)
z <- 100
x <- 1e4
eval_tidy(quo(x+y+z), tb)
## [1] 112 114 116
```

4 Anwendung

4.1 Zugriff auf Argumenttext

Eine Fragestellung zu Beginn des Kapitels war, wie `tibble()` Spaltennamen automatisch aus dem Text Argumente erstellen kann.

```
numbers <- 1:26
tibble(numbers, letters)
## # A tibble: 26 x 2
##    numbers letters
##      <int> <chr>
##  1       1 a
##  2       2 b
##  3       3 c
##  4       4 d
##  5       5 e
##  6       6 f
##  7       7 g
##  8       8 h
##  9       9 i
## 10      10 j
## # ... with 16 more rows
```

Wir können nun selbst eine Funktion mit dieser Fähigkeit schreiben.

```
capture_named <- function(...) {
  arg_list <- enquos(...)
  values <- lapply(arg_list, eval_tidy)
  arg_text <- sapply(arg_list, quo_text)
  names(values) <- make.names(arg_text) # make syntactically valid names
  return(values)
}
str(capture_named(numbers, letters, 1:3))
## List of 3
##  $ numbers: int [1:26] 1 2 3 4 5 6 7 8 9 10 ...
##  $ letters: chr [1:26] "a" "b" "c" "d" ...
##  $ X1.3   : int [1:3] 1 2 3
```

Dies klärt auch, wie `library(rlang)` ohne `"` funktionieren kann.

4.2 filter()

In `dplyr::filter()` können Spalten wie Variablen gebraucht werden.

```
tb <- tibble(x = 1:5, y = 5:1)
filter(tb, x > y)
## # A tibble: 2 x 2
##       x     y
##   <int> <int>
## 1     4     2
## 2     5     1
```

Wir erstellen eine vereinfachte Implementierung dieser Mechanik.

```
filter2 <- function(tb, fltr) {
  fltr <- enquo(fltr)
  rows <- eval_tidy(fltr, tb)
```

```
  tb[rows, ]
}
filter2(tb, x > y)
## # A tibble: 2 x 2
##       x     y
##   <int> <int>
## 1     4     2
## 2     5     1
```

4.3 source()

Mit der Funktion `source()` lassen sich andere Skript-Dateien in der aktuellen Umgebung ausführen.

Mit `rlang::parse_exprs()` können wir eine ähnliche Funktion schreiben.

```
source2 <- function(path) {
  file <- str_c(readLines(path, warn = FALSE), collapse = "\n")
  exprs <- parse_exprs(file)
  res <- NULL
  for (i in seq_along(exprs)) {
    res <- eval(exprs[[i]], caller_env())
  }
  res
}
```

5 Tidy-Eval-Framework

Einige Tidyverse-Funktionen nutzen **Non-Standard-Evaluation** (NSE), zB Tibble-Spalten wie Variablen benutzen.

Die NSE im Tidyverse basiert auf dem **Tidy-Eval-Framework**, welches von `rlang` bereitgestellt wird. Elemente dieses Frameworks sind Quosures, `eval_tidy()`, Forcing-Operatoren und Pronomen.

5.1 Forcing-Operatoren

Die Forcing-Operatoren sind:

- `!!` Bang-Bang
- `!!!` Big-Bang
- `{{}}` Curly-Curly (hier nicht besprochen)
- `:=` Walross (*walrus*)

Sie können nur in Argumenten von Funktionen genutzt werden, die das Tidy-Eval-Framework unterstützen. Dazu gehören einige Funktionen aus `rlang`, `lobstr` und dem Tidyverse, nicht jedoch Base-R-Funktionen.

```
a <- expr(f(x))
!!a # nicht ok, nicht in Argument
## Error in !a: invalid argument type
!!TRUE # doppelte Negation
## [1] TRUE
expr(1+!!a) # hier ok
## 1 + f(x)
```

Der **Bang-Bang**-Operator `!!` wertet den nachfolgenden Teilausdruck aus.

```
x <- 4
expr(1 + !!(2+3) + x + !!x)
## 1 + 5 + x + 4
```

`lobstr::ast()` interpretiert ihr Argument als Ausdruck. Mit `!!` zeigen wir den Syntaxbaum einer Ausdruck-Variable an.

```
a <- expr(f(y))
ast(a)
## a
ast(!!a)
## o-f
## \-y
ast(g(z, !!a))
## o-g
## +-z
## \-o-f
##   \-y
```

Der Operator `!!!` (**Big-Bang**) wertet zuerst den nachfolgenden Ausdruck aus; ist dieser ein Vektor, werden die Elemente als Liste aufgenommen.

```
x <- 1:3
str(exprs(!!x))
## List of 1
##  $ : int [1:3] 1 2 3
str(exprs(!!!x))
## List of 3
##  $ : int 1
##  $ : int 2
##  $ : int 3
```

Um einen Namen aus einer Variable als Argumentname zu verwenden, ersetzen wir `=` durch `:=` (**Walross**) und wenden `!!` auf die Variable an. Dabei verhält sich `:=` wie `=`, aber nur `:=` erlaubt `!!` auf der linken Seite.

```
name <- "asdf"
tibble(name = 1+2)
## # A tibble: 1 x 1
##    name
##   <dbl>
## 1     3
# tibble(!!name = 1+2) # ERROR
tibble(name := 1+2)
## # A tibble: 1 x 1
##    name
##   <dbl>
## 1     3
tibble(!!name := 1+2)
## # A tibble: 1 x 1
##    asdf
##   <dbl>
## 1     3
```

5.2 Pronomen

Pronomen (*pronouns*) sind die Namen `.data` und `.env`. Sie haben im Tidy-Eval-Framework eine besondere Bedeutung.

Sie können in Ausdrücken wie Tibbles behandelt werden. `.data` enthält die Namen der Data-Mask, `.env` die Umgebungsvariablen.

```
y <- 4
tb <- tibble(x = 1:3, y = 3:1)
eval_tidy(quo(y + y*10), tb)
## [1] 33 22 11
eval_tidy(quo(.data$y + .env$y*10), tb)
## [1] 43 42 41
```

5.3 Geschachtelte Quosures

`!!` wirkt auf Teil-Ausdrücke, die zu Quosures evaluieren, in besonderer Weise.

```
x <- 1
qua <- function(y) 10*y
q <- function() quo(x+2)
v <- function() qua(x+2)
identical(eval_tidy(quo(!!v())), v())
## [1] TRUE
identical(eval_tidy(quo(!!q())), q())
## [1] FALSE
identical(eval_tidy(quo(!!q())), x+2)
## [1] TRUE
```

`!!` gliedert den Inhalt eines Quosures in das äußere Quosure ein, wodurch ein viel tieferer Syntaxbaum entstehen kann. Bei der Eingliederung eines anderen Objektes mit `!!` entsteht nur ein Blatt im Syntaxbaum.

```
ast(7+!!v())
## o-`+`
## +-7
## \-30
ast(7+!!q())
## o-`+`
## +-7
## \-o-`+`
##   +-x
##   \-2
```

Quosures können Teil-Quosures mit unterschiedlichen Umgebungen haben. `eval_tidy()` wertet jedes Quosure in seiner Umgebung aus.

```
y <- 1
create_q <- function() {
  y <- 10
  quo(y)
}
create_qu <- function(x) {
  y <- 100
  quo(y + !!x)
}
qu <- create_qu(create_q())
qu # y with different environments
```

```
## <quosure>
## expr: ^y + (^y)
## env:  0000000020A89000
eval_tidy(qu)
## [1] 110
eval_tidy(qu, list(y = 1e3))
## [1] 2000

# vergleiche mit:
quo(y + y) # y from same environment
## <quosure>
## expr: ^y + y
## env:  global
eval_tidy(quo(y + y))
## [1] 2
eval_tidy(quo(y + y), list(y = 1e3))
## [1] 2000
```

Beachte die Ausgabe der Quosures als `^y + (^y)` bzw `^y + y`. Das Symbol `^` markiert den Beginn eines neuen Teil-Quosures, welches eine eigene Umgebung hat.

Im folgenden Beispiel wollen wir `tidy_eval(XXX == 1, tb)` auswerten, wobei wir `XXX` durch einen Namen einer Spalte von `tb` ersetzen. Dieser Name soll jedoch nicht fest vorgegeben, sondern in einer Variable gespeichert sein.

```
y <- 4
tb <- tibble(x = 1:5, y = 5:1)

# verschiedene Ansätze Namen zu speichern
var <- y # normale Variable
q <- quo(y) # Quosure

# Versuch 1
qu <- quo(var == 1)
qu
## <quosure>
## expr: ^var == 1
## env:  global
eval_tidy(qu, tb) # nein
## [1] FALSE

# Versuch 2
qu <- quo(q == 1)
qu
## <quosure>
## expr: ^q == 1
## env:  global
eval_tidy(qu, tb) # nein
## Error: Base operators are not defined for quosures.
## Do you need to unquote the quosure?
##
##   # Bad:
##   myquosure == rhs
##
##   # Good:
```

```
##   !!myquosure == rhs

# Versuch 3
qu <- quo(!!q == 1)
qu
## <quosure>
## expr: ^(^y) == 1
## env:  global
eval_tidy(qu, tb) # ja :)
## [1] FALSE FALSE FALSE FALSE  TRUE
```

Bemerkung: Ist die Variable als String gespeichert, können wir sie in ein Quosure umwandeln und dann wie oben verfahren. Einfacher geht es jedoch mit dem Pronomen `.data`.

```
str <- "y" # String

qu <- quo(!!parse_quo(str, current_env()) == 1)
qu
## <quosure>
## expr: ^(^y) == 1
## env:  global
eval_tidy(qu, tb)
## [1] FALSE FALSE FALSE FALSE  TRUE

qu <- quo(.data[[str]] == 1)
qu
## <quosure>
## expr: ^.data[["y"]] == 1
## env:  global
eval_tidy(qu, tb)
## [1] FALSE FALSE FALSE FALSE  TRUE
```

Der große Vorteil der Nutzung von Quosures anstatt Strings ist die Möglichkeit beliebige, syntaktisch korrekte Ausdrücke als Argument zu geben.

```
q <- quo(y %% 2)
qu <- quo(!!q == 1)
qu
## <quosure>
## expr: ^(^y %% 2) == 1
## env:  global
eval_tidy(qu, tb)
## [1]  TRUE FALSE  TRUE FALSE  TRUE
```

Jetzt setzen wir obiges Vorgehen analog mit Funktionsargumenten um. Dabei ersetzen wir `quo()` durch `enquo()`. Mit dieser Technik schreiben wir eine Funktion, die `filter2(tb, XXX == 1)` aufruft, wobei `XXX` ein Ausdruck ist, der Spalten aus `tb` wie Variablen benutzt.

```
y <- 4
tb <- tibble(x = 1:5, y = 5:1)
filter2 <- function(tb, fltr) {
  qu <- enquo(fltr)
  rows <- eval_tidy(qu, tb)
  tb[rows, ]
}
```

```r
# Versuch 1
filter2x1 <- function(var) {
  filter2(tb, var == 1)
}
filter2x1(y) # nein
## # A tibble: 0 x 2
## # ... with 2 variables: x <int>, y <int>

# Versuch 2
filter2x2 <- function(var) {
  q <- enquo(var)
  filter2(tb, q == 1)
}
filter2x2(y) # nein
## Error: Base operators are not defined for quosures.
## Do you need to unquote the quosure?
##
##   # Bad:
##   myquosure == rhs
##
##   # Good:
##   !!myquosure == rhs

# Versuch 3
filter2x3 <- function(var) {
  q <- enquo(var)
  filter2(tb, !!q == 1)
}
filter2x3(y) # ja :)
## # A tibble: 1 x 2
##       x     y
##   <int> <int>
## 1     5     1
filter2x3(y %% 2)
## # A tibble: 3 x 2
##       x     y
##   <int> <int>
## 1     1     5
## 2     3     3
## 3     5     1
```

5.4 Anwendung im Tidyverse

Argumente von Funktionen im Tidyverse, bei denen Spalten wie Variablen benutzt werden können, unterstützen das Tidy-Eval-Framework.

Im Folgenden zeigen wir ein paar Anwendungsbeispiele

Grouping-Variable als Argument

```r
df <- tibble(
  g1 = c(1, 1, 2, 2, 2),
  g2 = c(1, 2, 1, 2, 1),
  a = sample(5),
  b = sample(5)
```

```
)

# Versuch 1
my_summarise <- function(df, group_var) {
  df %>%
    group_by(group_var) %>%
    summarise(a = mean(a))
}
my_summarise(df, g1) # nein
## Error: Must group by variables found in `.data`.
## * Column `group_var` is not found.

# Versuch 2
my_summarise <- function(df, group_var) {
  q <- enquo(group_var)
  df %>%
    group_by(!!q) %>%
    summarise(a = mean(a))
}
my_summarise(df, g1) # ja :)
## # A tibble: 2 x 2
##      g1     a
##   <dbl> <dbl>
## 1     1  4
## 2     2  2.33
```

Zusammenzufassende Variable als Argument

```
my_summarise2 <- function(df, expr) {
  q <- enquo(expr)
  summarise(df,
    mean = mean(!!q),
    sum = sum(!!q),
    n = n()
  )
}
my_summarise2(df, a)
## # A tibble: 1 x 3
##    mean   sum     n
##   <dbl> <int> <int>
## 1     3    15     5
my_summarise2(df, a * b)
## # A tibble: 1 x 3
##    mean   sum     n
##   <dbl> <int> <int>
## 1   8.6    43     5
```

Name neuer Spalten aus Variable

`quo_name()` wandelt ein Quosure in den Text ihres Ausdruck um.

```
my_mutate <- function(df, expr) {
  q <- enquo(expr)
  mean_name <- str_c("mean_", quo_name(q))
  sum_name <- str_c("sum_", quo_name(q))
```

```
  mutate(df,
    !!mean_name := mean(!!q),
    !!sum_name := sum(!!q)
  )
}
my_mutate(df, a)
## # A tibble: 5 x 6
##      g1    g2     a     b mean_a sum_a
##   <dbl> <dbl> <int> <int>  <dbl> <int>
## 1     1     1     5     4      3    15
## 2     1     2     3     2      3    15
## 3     2     1     4     1      3    15
## 4     2     2     1     3      3    15
## 5     2     1     2     5      3    15
```

Beliebige Anzahl an Grouping-Variablen

```
my_summarise <- function(df, ...) {
  qs <- enquos(...)
  df %>%
    group_by(!!!qs) %>%
    summarise(a = mean(a))
}
my_summarise(df, g1, g2)
## `summarise()` has grouped output by 'g1'. You can override using the `.groups` argument.
## # A tibble: 4 x 3
## # Groups:   g1 [2]
##      g1    g2     a
##   <dbl> <dbl> <dbl>
## 1     1     1     5
## 2     1     2     3
## 3     2     1     3
## 4     2     2     1
```