Métaprogrammation en R

Sophie Baillargeon, Université Laval

2019-04-01

Table des matières

La métaprogrammation se définit par **l'écriture d'un programme qui écrit lui-même un programme**. Par exemple, en R, il est possible d'écrire un bout de code qui compose d'abord une ou des instructions sous forme de chaînes de caractères ou d'expressions, puis qui évalue ces instructions dans un deuxième temps. La métaprogrammation est typiquement utilisée pour rendre un programme plus succinct, donc potentiellement plus facile à comprendre, ou pour automatiser certaines tâches.

Les sections qui suivent décrivent quelques outils en R pour manipuler des éléments de langage sans les évaluer, ce qui permet de réaliser de la métaprogrammation.

Assignation d'une valeur à un nom d'objet avec `assign`

La façon usuelle d'écrire une assignation en R est la suivante.

```
obj <- 1
```

Dans cette instruction, `1` est la valeur à assigner et `obj` est le nom de l'objet qui contiendra la valeur.

Mais comment assigner une valeur à un nom d'objet que nous souhaitons manipuler sous forme de chaîne de caractères ? Par exemple, supposons que ce nom est stocké dans l'objet `nom`.

```
nom <- "nomObjet"
```

Nous voulons assigner une valeur à un objet qui portera le nom stocké dans `nom`, peu importe la chaîne de caractère que `nom` contient. Cette assignation peut être réalisée avec la fonction `assign` comme suit :

```
assign(x = nom, value = 1)
```

Que contient notre environnement de travail maintenant ?

```
ls()
```

```
## [1] "nom"       "nomObjet" "obj"
```

Nous y trouvons maintenant un objet nommé `nomObjet`, ce qui correspond à la chaîne de caractères stockée dans `nom`. Cet objet contient la valeur `1`.

```
nomObjet
```

```
## [1] 1
```

Voici un autre exemple d'utilisation de la fonction `assign`. Supposons que nous souhaitons créer 5 objets, nommés `obj1` à `obj5`. Ces objets doivent contenir un vecteur d'entiers allant de 1 à `x` où `x` est le numéro de l'objet.

```
for (i in 1:5) {
  assign(x = paste0("obj", i), value = 1:i)
}
```

Vérifions que ces objets ont bien été créés dans notre environnement de travail.

```
ls()
```

```
## [1] "i"        "nom"      "nomObjet" "obj"      "obj1"     "obj2"
## [7] "obj3"     "obj4"     "obj5"
```

Retour de la valeur assignée à un nom d'objet avec `get`

Pour atteindre la valeur assignée à un nom d'objet, nous sommes habitués à passer directement par ce nom, comme dans cet exemple.

```
str(nomObjet)
```

```
##  num 1
```

Mais comment procéder avec un nom sous forme de chaîne de caractère ? Il faut utiliser la fonction `get` comme suit.

```
str(get(nom))
```

```
##  num 1
```

Ce qui ne retourne pas la même chose que ceci.

```
str(nom)
```

```
# Équivalent à
str("nomObjet")
```

```
##  chr "nomObjet"
```

Par exemple, pour afficher le contenu des objets nommés `obj1` à `obj5`, nous pouvons procéder comme suit.

```r
for (i in 1:5) {
  cat(paste0("obj", i, " ="), get(paste0("obj", i)), "\n")
}
```

```
## obj1 = 1
## obj2 = 1 2
## obj3 = 1 2 3
## obj4 = 1 2 3 4
## obj5 = 1 2 3 4 5
```

Appel d'une fonction avec `do.call`

Nous venons d'apprendre comment manipuler un nom d'objet sous forme de chaîne de caractère. Comment procéder lorsque ce nom est celui d'une fonction que nous souhaitons appeler ?

Il est alors possible d'utiliser `get`, mais une autre fonction peut aussi nous être utile : `do.call`.

Par exemple, les trois instructions suivantes provoquent toute l'évaluation du même appel à la fonction `median`.

```r
median(x = 1:10)
```

```
## [1] 5.5
```

```r
get("median")(x = 1:10)
```

```
## [1] 5.5
```

```r
do.call("median", args = list(x = 1:10))
```

```
## [1] 5.5
```

En fait, l'avantage de `do.call` n'est pas qu'il soit capable de manipuler une fonction dont le nom est sous forme de chaîne de caractères. D'ailleurs, `do.call` accepte comme premier argument la fonction directement. Sa principale utilité est plutôt d'accepter sous forme de liste les arguments à inclure dans l'appel à une fonction. Cette liste peut être construite par étapes, potentiellement conditionnelles.

Voici un exemple de fonction qui exploite le potentiel de la fonction `do.call`.

```r
#' Calcul de statistiques descriptives
#'
#' Cette fonction permet de calculer des statistiques descriptives au choix
#'
#' @param x vecteur d'observations
#' @param choix une chaîne de caractères spécifiant le nom de la fonction
#'              à appeler pour le calcul, soit "table" (par défaut), "mean",
#'              "median", "sd" ou "mad"
#' @param retirerNA un logique spécifiant si les valeurs manquantes (NA)
#'                  doivent être retirées avant le calcul (par défaut TRUE)
#'
#' @return le résultat de l'appel à la fonction choisie
#'
choixstat <- function(x, choix = c("table", "mean", "median", "sd", "mad"),
                      retirerNA = TRUE) {
  choix <- match.arg(choix)
  arguments <- list(x)
  if (choix == "table"){
    arguments$useNA <- if (retirerNA) "no" else "ifany"
```

```
  } else {
    arguments$na.rm <- retirerNA
  }
  do.call(what = choix, args = arguments)
}
```

```
choixstat(x = c(2,3,2,3,3,4,NA,3), choix = "median", retirerNA = TRUE)
```

```
# Équivalent
median(c(2,3,2,3,3,4,NA,3), na.rm = TRUE)
```

```
## [1] 3
```

```
choixstat(x = c(2,3,2,3,3,4,NA,3), choix = "median", retirerNA = FALSE)
```

```
# Équivalent à
median(c(2,3,2,3,3,4,NA,3), na.rm = FALSE)
```

```
## [1] NA
```

```
choixstat(x = c(2,3,2,3,3,4,NA,3), choix = "table", retirerNA = TRUE)
```

```
# Équivalent à
table(c(2,3,2,3,3,4,NA,3), useNA = "no")
```

```
##
## 2 3 4
## 2 4 1
```

```
choixstat(x = c(2,3,2,3,3,4,NA,3), choix = "table", retirerNA = FALSE)
```

```
# Équivalent à
table(c(2,3,2,3,3,4,NA,3), useNA = "ifany")
```

```
##
##    2    3    4 <NA>
##    2    4    1    1
```

Manipulation de formules

Les formules sont des éléments de langage R particuliers. Ils servent à spécifier des modèles statistiques. L'instruction suivante est un exemple de création de formule en R.

```
f1 <- y ~ x1 + x2
```

Une formule contient un opérateur `~`, possiblement avec un argument à gauche pour spécifier la ou les variables réponses du modèle et un argument à droite pour spécifier la ou les variables explicatives du modèle. Des informations sur les formules ont été fournies dans le cours sur les calculs statistiques et mathématiques en R.

R reconnaît que `f1` est bien une formule. Il la décompose même en trois parties : l'opérateur `~`, la partie de gauche du modèle (en anglais *LHS* pour *left hand side*) et la partie de droite du modèle (en anglais *RHS* pour *right hand side*).

```
str(f1)
```

```
## Class 'formula'  language y ~ x1 + x2
##   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

```
f1[1]
```

```
## `~`()
```

```
f1[2]
```

```
## y()
```

```
f1[3]
```

```
## (x1 + x2)()
```

Fonction `as.formula`

La fonction `as.formula` permet de créer une formule à partir d'une chaîne de caractères.

```
str("y ~ x1 + x2")
```

```
##  chr "y ~ x1 + x2"
```

```
f2 <- as.formula("y ~ x1 + x2")
str(f2)
```

```
## Class 'formula'  language y ~ x1 + x2
##   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

Cette possibilité de transformer une chaîne de caractères en formule s'avère pratique, par exemple, lorsque nous avons besoin de construire une formule comprenant un grand nombre de termes identifiables de façon automatique.

Voici un exemple de fonction qui utilise `as.formula` pour construire un modèle de régression polynomial.

```
#' Régression polynomiale
#'
#' Ajustement d'un modèle de régression polynomial entre deux variables
#'
#' @param y vecteur des observations de la variable réponse
#' @param x vecteur des observations de la variable explicative
#' @param dg degré du modèle polynomial à ajuster
#'
#' @return sortie de la fonction lm pour le modèle demandé
#'
regPoly <- function(y, x, dg){
  formule <- paste("y ~",
                   paste(paste0("I(x^", 1:dg, ")"), collapse = " + ")
                   )
  lm(as.formula(formule))
}
```

```
regPoly(y = cars$dist, x = cars$speed, dg = 3)
```

```
##
## Call:
## lm(formula = as.formula(formule))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)
##   -19.50505      6.80111     -0.34966      0.01025
```

```
regPoly(y = cars$dist, x = cars$speed, dg = 5)
```

```
##
## Call:
## lm(formula = as.formula(formule))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)       I(x^4)       I(x^5)
##   -2.650053     5.484259    -1.426123     0.194049    -0.010040     0.000179
```

Méthode `update.formula`

La méthode `update.formula` permet de partir d'un modèle et de le modifier. Par exemple, reprenons la formule `f1`.

```
f1
```

```
## y ~ x1 + x2
```

Ajoutons-y une variable explicative.

```
update(f1, . ~ . + x3)
```

```
## y ~ x1 + x2 + x3
```

Ou encore, retirons une variable.

```
update(f1, . ~ . - x2)
```

```
## y ~ x1
```

Nous pourrions aussi transformer une variable.

```
update(f1, sqrt(.) ~ .)
```

```
## sqrt(y) ~ x1 + x2
```

La fonction `update` est générique. Si le premier argument qu'elle reçoit est une formule, elle appelle la méthode `update.formula`. Dans un appel à `update.formula`, un `.` représente une partie de la formule originale. Le `.` à gauche du `~` représente le LHS du modèle fourni comme premier argument et le `.` à droite du `~` le RHS du modèle fourni comme premier argument.

Manipulation d'instructions

Il est possible d'écrire une instruction R complète, sous forme de chaîne de caractères ou d'expression R, sans l'évaluer.

Prenons par exemple l'instruction R suivante : `median(x = 1:10)`. Si nous soumettons cette instruction dans la console, elle sera évaluée et son résultat sera affiché.

```
median(x = 1:10)
```

```
## [1] 5.5
```

Si nous ajoutons une assignation au début de l'instruction comme suit

```
out <- median(x = 1:10)
```

le résultat ne sera pas affiché, mais l'objet `out` contiendra le résultat de l'évaluation de `median(x = 1:10)`.

```
out
```

```
## [1] 5.5
```

Mais comment stocker dans un objet l'instruction elle-même ?

Fonction `quote`

La fonction `quote` retourne une instruction R non évaluée, que nous pouvons appeler « expression ».

```
out_quote <- quote(median(x = 1:10))
out_quote
```

```
## median(x = 1:10)
```

```
str(out_quote)
```

```
##  language median(x = 1:10)
```

C'est un « élément de langage ».

Fonction `eval`

Lorsque nous désirons évaluer une expression R, nous pouvons la fournir en entrée à la fonction `eval`.

```
eval(out_quote)
```

```
## [1] 5.5
```

Fonction `call`

Si l'instruction que nous souhaitons manipuler est seulement un appel à une fonction, nous pouvons aussi créer l'expression non évaluée de l'instruction avec la fonction `call`.

```
out_call <- call("median", x = 1:10)
out_call
```

```
## median(x = 1:10)
```

```
str(out_call)
```

```
##  language median(x = 1:10)
```

Les objets `out_call` et `out_quote` sont équivalents.

```
all.equal(out_quote, out_call)
```

```
## [1] TRUE
```

Évaluer `out_call` se réalise de la même façon qu'évaluer `out_quote`.

```
eval(out_call)
```

```
## [1] 5.5
```

Fonction `parse`

Maintenant, ce qui serait vraiment utile est de pouvoir transformer une chaîne de caractères en expression C'est ce que nous permet de faire la fonction `parse`.

Par exemple, si nous avons la chaîne de caractère suivante

```r
instruc_car <- "median(x = 1:10)"
str(instruc_car)
```

```
##  chr "median(x = 1:10)"
```

nous pouvons la transformer en expression non évaluée avec `parse` comme suit.

```r
out_parse <- parse(text = instruc_car)
out_parse
```

```
## expression(median(x = 1:10))
```

L'objet `out_parse` n'a pas tout à fait la même structure que `out_quote` ou `out_call`, mais il s'évalue de la même façon, avec la fonction `eval`.

```r
eval(out_parse)
```

```
## [1] 5.5
```

Nous avons donc parcouru le chemin de transformation suivant :

> instruction sous forme de chaîne de caractères,
> > expression non évaluée avec `parse`,
> > évaluation de l'expression avec `eval`.

Grâce à ces outils, nous pourrions améliorer la fonction `regPoly`. Vous aurez peut-être remarqué que dans l'impression de la sortie de cette fonction, le `Call` a toujours la même allure : `lm(formula = as.formula(formule))`. Ce n'est pas très informatif. Voici une deuxième version de cette fonction, utilisant `parse` et `eval` plutôt que `as.formula`, qui produit un affichage amélioré.

```r
regPoly2 <- function(y, x, dg){
  instruc <- paste0("lm(y ~ ",
                    paste(paste0("I(x^", 1:dg, ")"), collapse = " + "),
                    ")")
  eval(parse(text = instruc))
}
```

```r
regPoly2(y = cars$dist, x = cars$speed, dg = 3)
```

```
##
## Call:
## lm(formula = y ~ I(x^1) + I(x^2) + I(x^3))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)
##   -19.50505      6.80111     -0.34966      0.01025
```

```r
regPoly2(y = cars$dist, x = cars$speed, dg = 5)
```

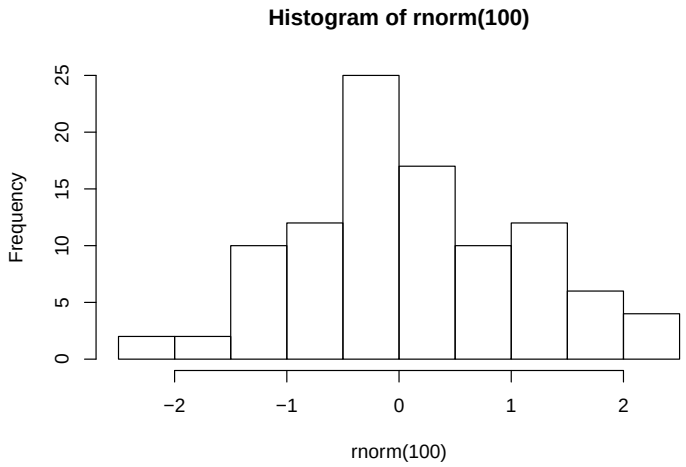
```
##
## Call:
## lm(formula = y ~ I(x^1) + I(x^2) + I(x^3) + I(x^4) + I(x^5))
##
## Coefficients:
## (Intercept)       I(x^1)       I(x^2)       I(x^3)       I(x^4)
##   -2.650053     5.484259    -1.426123     0.194049    -0.010040
##      I(x^5)
##    0.000179
```

Manipulation de l'appel d'une fonction

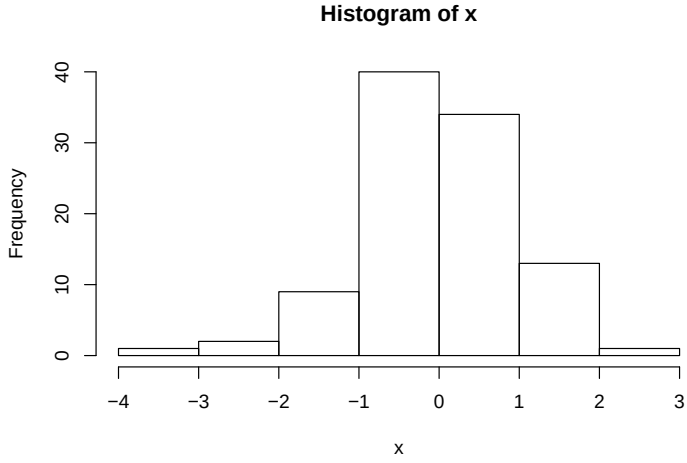
Dans le corps d'une fonction, il est parfois utile de pouvoir manipuler l'appel de la fonction.

Par exemple, la fonction `hist` utilise la façon dont la fonction a été appelée pour écrire le titre du graphique et le nom de l'axe des x.

```
hist(rnorm(100))
```




```
x <- rnorm(100)
hist(x)
```



Comment arriver à faire pareil ?

Fonctions `substitute` et `deparse`

Essayons de trouver dans le corps de la fonction `hist` comment elle procède. En fait, la fonction `hist` étant générique, allons voir dans le corps de sa méthode par défaut.

```
View(hist.default)
# résultat non affiché ici, soumettre la commande dans la console pour le voir
```

L'instruction `deparse(substitute(x))` est utilisée pour créer l'objet `xname`, qui est ensuite utilisé dans les

valeurs par défaut des arguments `main` et `xlab`.

Exemples pour mieux comprendre `substitute` et `deparse` :

```
fct1 <- function(x){
  x
}
test <- fct1(1:10)
test
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
str(test)
```

```
##  int [1:10] 1 2 3 4 5 6 7 8 9 10
```

La fonction `fct1` retourne un vecteur

```
fct2 <- function(x){
  substitute(x)
}
test <- fct2(1:10)
test
```

```
## 1:10
```

```
str(test)
```

```
##  language 1:10
```

La fonction `fct2` retourne une expression non évaluée (comme `quote` ou `call`).

```
fct3 <- function(x){
  deparse(substitute(x))
}
test <- fct3(1:10)
test
```

```
## [1] "1:10"
```

```
str(test)
```

```
##  chr "1:10"
```

Dans le corps de la fonction `fct3`, l'expression non évaluée est transformée en chaîne de caractères par la fonction `deparse`.

Ainsi, la fonction `deparse` permet de faire l'inverse de la fonction `parse` :

- `deparse` : expression → chaîne de caractères,
- `parse` : chaîne de caractères → expression.

Fonction `match.call`

Avec `substitute`, nous accédons à une instruction fournie pour un argument. La fonction `match.call`, que nous avons déjà mentionnée dans les notes sur les fonctions, permet quant à elle d'accéder à l'appel complet de la fonction.

```
fct4 <- function(x){
  match.call()
}
```

```r
test <- fct4(1:10)
test
```

```
## fct4(x = 1:10)
```

```r
str(test)
```

```
##  language fct4(x = 1:10)
```

Elle retourne une expression non évaluée, comme `substitute`. Dans cette expression, il est possible d'accéder à des arguments en particulier.

```r
fct5 <- function(x, y){
  appel <- match.call()
  list(appel_complet = appel,
       arg_x_exp = appel$x,
       arg_y_car = deparse(appel$x),
       arg_y_exp = appel$y)
}
test <- fct5(x = 1:10, y = "a")
str(test)
```

```
## List of 4
##  $ appel_complet: language fct5(x = 1:10, y = "a")
##  $ arg_x_exp    : language 1:10
##  $ arg_y_car    : chr "1:10"
##  $ arg_y_exp    : chr "a"
```

Dans cet exemple, la valeur fournie à l'argument `y` ne contient pas un appel à une fonction ou un opérateur. Il s'agit d'un seul élément, de type caractère. Son expression non évaluée est donc la valeur en caractères elle-même.

Synthèse

Manipuler des éléments de langage sans les évaluer

- assignation d'une valeur à un nom : `assign` ;
- retour de la valeur assignée à un nom : `get` ;
- appel d'une fonction : `do.call` ;
- manipulation de formules : `as.formula` et `update.formula` ;
- manipulation d'instructions : `quote`, `eval`, `call` et `parse` ;
- manipulation de l'appel d'une fonction : `substitute`, `deparse` et `match.call`.

Références

- R Core Team (2018). *R Language Definition*. R Foundation for Statistical Computing. Chapitre 6. URL http://cran.r-project.org/doc/manuals/r-devel/R-lang.html#Computing-on-the-language
- Wickham, H. (2014). *Advanced R*. CRC Press.
  - Édition 1, Chapitres 13-14. URLs http://adv-r.had.co.nz/Computing-on-the-language.html, http://adv-r.had.co.nz/Expressions.html
  - Édition 2, Section 4. URL https://adv-r.hadley.nz/metaprogramming.html