# svTidy et le 'formula-masking'

## Une alternative intéressante à {dplyr} et {tidyr} pour remanier vos jeux de données

Philippe Grosjean & Guyliann Engels (Écologie numérique, Complexys & Infortech, Université de Mons, BE)


**tl;dr**

{svTidy} reimplémente les fonctions de {dplyr} et {tidyr}, mais utilise des formules pour les évaluations non standard. Cela présente de nombreux avantages. Les plus importants sont :

- indication explicite de l'utilisateur qu'une évaluation non standard est souhaitée
- généralisation/inclusion du code dans une fonction facilitées

Le code de {svTidy} est également souvent plus rapide.

*{svTidy} sur GitHub https://github.com/SciViews/svTidy/ et R-Universe https://sciviews.r-universe.dev/. Aussi https://www.sciviews.org/svTidy/.*

## Évaluation non standard (NSE)

R permet de retravailler le code fourni aux arguments d'une fonction pour l'évaluer de manière différente. C'est l'**évaluation non standard** (NSE par opposition à SE, évaluation standard). Par exemple, pour extraire une colonne d'un data frame, on peut faire en R de base (évaluation standard) :

```
starwars[["species"]] # SE
```

Une version plus courte utilise l'opérateur `$`. Cet opérateur évalue son second argument (ici `species`) de manière *non standard* de telle sorte que les guillemets sont optionnels, malgré qu'il s'agisse bien ici d'une chaîne de caractères :

```
starwars$species  # NSE pour species (guillemets optionnels)
```

Le Tidyverse implémente sa propre version de NSE avec la 'tidy evaluation'. Un code équivalent s'écrit avec {dplyr} comme ceci :

```
pull(starwars, species) # NSE pour species (tidy evaluation)
```

Dans le cas de `pull()` le NSE est identique à `$`, mais en réalité la 'tidy evaluation' permet des constructions bien plus complexes.

## NSE avec le 'formula-masking'

Il n'est pas toujours facile de déterminer quand un argument est évalué SE ou NSE. Et quand il est NSE, avec quel mécanisme.

{svTidy} introduit le **'formula-masking'**, une approche NSE *explicite* et *standardisée*. Les formules de R contiennent toutes l'opérateur 'tilde' ~ qui ne sert à rien d'autre, et est donc facile à identifier dans le code.

En {dplyr}, on écrit :

```
# {dplyr} version
ages_dplyr <-
  starwars |>
  filter(species == "Human") |>
  mutate(age = birth_year + 2) |>
  summarise(
    mean_age = mean(age, na.rm = TRUE),
    sd_age   = sd(age, na.rm = TRUE),
    n_age    = sum(!is.na(age)),
    .by      = "gender"
  )
```

En {svTidy} les sections NSE sont plus explicites (précédées du ~) :

```
# {svTidy} version
ages_svTidy <-
  starwars |>
  filter_(~species == "Human") |>
  mutate_(age = ~birth_year + 2) |>
  summarise_(
    mean_age = ~mean(age, na.rm = TRUE),
    sd_age   = ~sd(age, na.rm = TRUE),
    n_age    = ~sum(!is.na(age)),
    .by      = "gender"
  )
}
```

*Notez aussi que les fonctions {svTidy} portent le même nom que celles de {dplyr}, mais terminées par un _.*

## Inclusion dans une fonction

La 'tidy evaluation' est assortie de mécanismes complexes pour permettre la *généralisation* du code et son inclusion dans une fonction. Le 'formula-masking' permet un traitement plus direct et intuitif.

```
# {dplyr} version
f_dplyr <- function(data, subset, var, year, group) {
  filter(data, {{ subset }}) |>
    mutate({{var}} := .data$birth_year + .env$year) |>
    summarise(
      "mean_{{var}}" := mean({{ var }}, na.rm = TRUE),
      .by = {{ group }})
}
```

```
# {svTidy} version
f_svTidy <- function(data, subset, var, year, group) {
  filter_(data, subset) |>
    mutate_(var ~ birth_year + year) |>
    summarise_(
      "mean_{{var}}" ~ mean(var, na.rm = TRUE),
      .by = group)
}
```

## Bonus : code optimisé - vitesse et mémoire

| expression | median | itr/sec | mem_alloc |
|---|---|---|---|
| f_dplyr | 853.4µs | 1154 | 2.37MB |
| f_svTidy | 98.7µs | 10065 | 737.68KB |

Détails, y compris sur le benchmark, ici :