Introduction à la théorie des langages de programmation

Gilles Dowek & Jean-Jacques Lévy

Les auteurs tiennent à remercier Gérard Assayag, Antonio Bucciarelli, Roberto Di Cosmo, Xavier Leroy, Dave MacQueen, Luc Maranget, Michel Mauny, François Pottier, Didier Rémy, Alan Schmitt, Élodie-Jane Sims et Véronique Viguié Donzeau-Gouge.

Qu'est-ce que la théorie des langages de programmation ?

Nous sommes encore très loin d'avoir trouvé un langage de programmation définitif. Presque chaque jour, de nouveaux langages sont créés et de nouvelles fonctionnalités sont ajoutées aux langages anciens. Améliorer les langages de programmation permet de rendre les programmes plus sûrs, plus rapides à développer et plus faciles à maintenir. Cela permet aussi de répondre à des besoins nouveaux, comme la conception de programmes parallèles, distribués ou mobiles.

La première chose que l'on doit décrire, quand on définit un langage de programmation, est sa *syntaxe*. Faut-il écrire `x := 1` ou `x = 1` ? Faut-il mettre des parenthèses après un `if` ou non ? Plus généralement, quelles sont les suites de symboles qui forment un programme ? On dispose pour cela d'un outil efficace : la notion de *grammaire formelle*. À l'aide d'une grammaire, on peut décrire la syntaxe d'un langage de manière qui ne laisse pas de place à l'interprétation et qui rende possible l'écriture d'un programme qui reconnaît les programmes syntaxiquement corrects.

Mais savoir ce qu'est un programme syntaxiquement correct ne suffit pas pour savoir ce qui se passe quand on exécute un tel programme. Quand on définit un langage de programmation, on doit donc également décrire sa *sémantique*, c'est-à-dire ce qui se passe quand on exécute un programme. Deux langages peuvent avoir la même syntaxe mais des sémantiques différentes.

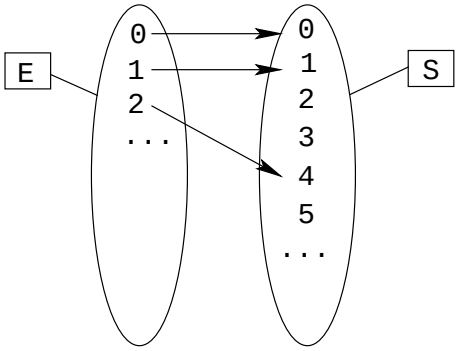
Par exemple, on explique souvent ainsi ce qui se passe quand on évalue une fonction : *« Le résultat* `V` *de l'évaluation d'une expression de la forme* `f` $\mathtt{e}_1$ ... $\mathtt{e}_n$*, où le symbole* `f` *est une fonction définie par l'expression* `f` $\mathtt{x}_1$ ... $\mathtt{x}_n$ `= e'` *est obtenu ainsi. On évalue les arguments* $\mathtt{e}_1$*, ...,* $\mathtt{e}_n$*, ce qui donne les valeurs* $\mathtt{W}_1$*, ...,* $\mathtt{W}_n$*, on lie ensuite ces valeurs aux variables* $\mathtt{x}_1$*, ...,* $\mathtt{x}_n$ *et on évalue enfin l'expression* `e'`*. La valeur* `V` *est le résultat de cette évaluation. »* Cette explication verbale — c'est-à-dire exprimée dans une langue naturelle comme le français — de la sémantique du langage permet de comprendre ce qui se passe quand on exécute un programme. Mais est-elle absolument rigoureuse ? Considérons, par exemple, le programme

```
f x y = x
g z = (n = n + z; n)
n = 0; print(f (g 2) (g 7))
```

Selon la manière dont on interprète l'explication verbale ci-dessus, on peut déduire que ce programme affichera `2` ou qu'il affichera `9`. En effet, cette explication verbale ne nous a pas indiqué s'il fallait évaluer `g 2` avant `g 7` ou après et l'ordre d'évaluation importe dans ce cas. Il aurait donc fallu écrire « On évalue les arguments e_1, ..., e_n *en commençant par* e_1 » ou alors *« en commençant par* e_n *»*. En lisant une explication ambiguë, deux programmeurs peuvent comprendre des choses différentes. Plus grave : les concepteurs de deux compilateurs du même langage peuvent choisir des conventions différentes. Le même programme donnera alors des résultats différents selon qu'il est compilé par un compilateur ou par l'autre.

Nous savons que le français est trop imprécis pour exprimer la syntaxe d'un langage et qu'il vaut mieux utiliser un langage formel pour cela. Le français est également trop imprécis pour exprimer la sémantique d'un langage de programmation et, ici aussi, il vaut mieux utiliser un langage formel.

Qu'est-ce que la sémantique d'un programme ? Prenons l'exemple d'un programme `p` qui demande un entier à l'utilisateur, l'élève au carré et affiche le résultat de cette opération. Décrire ce que fait ce programme consiste à décrire une relation `R` qui relie chaque valeur d'entrée à la valeur de sortie correspondante.



La sémantique de ce programme est donc une relation `R` reliant des éléments de l'ensemble `E` des valeurs d'entrées et des éléments de l'ensemble `S` des valeurs de sorties, c'est-à-dire une partie de l'ensemble $\texttt{E} \times \texttt{S}$.

La sémantique d'un programme est donc une relation à deux places. La sémantique d'un langage de programmation, quant à elle, est une relation à trois places : « le programme `p` sur la valeur d'entrée `e` donne la valeur de sortie `s` ». On note cette relation $\texttt{p, e} \hookrightarrow \texttt{s}$. Le programme `p` et l'entrée `e` sont ce dont on dispose avant le début de l'exécution du programme. Bien souvent, on agrège ces deux informations en un *terme* `p e` auquel la sémantique du langage attribue une valeur. La sémantique d'un langage est alors une relation à deux places $\texttt{t} \hookrightarrow \texttt{s}$.

Un langage permettant d'exprimer la sémantique des langages de programmation est donc un langage permettant de définir des relations.

Quand la sémantique d'un programme est une relation fonctionnelle, c'est-à-dire que, pour une valeur d'entrée, il y a au plus une valeur de sortie, on

dit que le programme est *déterministe*. Les jeux vidéos sont des exemples de programmes non déterministes, car il faut un peu d'aléa pour qu'un jeu vidéo soit amusant. Un langage est déterministe si tous les programmes écrits dans ce langage le sont ou, de manière équivalente, si sa sémantique est une relation fonctionnelle. Dans ce cas, on peut définir sa sémantique en utilisant, non un langage de définition de relations, mais un langage de définition de fonctions.