Analyse syntaxique et Sémantique Opérationnelle Structurée

7 novembre 2018

1 Définition et analyse d'un langage de programmation simple

Comme en While, on considère qu'un programme est :
— soit ne rien faire,
— soit une affectation (d'une expression à une variable),
— soit deux programmes mis bout-à-bout (séquence),
— soit une instruction conditionnelle (constituée d'une expression, d'un programme à exécuter si l'expression vaut 1, et d'un second programme à exécuter si l'expression vaut 0),
— soit une boucle while (constitué d'une expression et d'un corps ; la condition d'arrêt étant que l'expression vaut 0).

De plus, par souci de simplification, on considérera ici :
— que toutes les variables sont booléennes (et valent 0 ou 1)
— que la condition d'un if ou d'un while est toujours constituée d'une variable seulement
— que le membre droit d'une affectation peut être : soit 0, soit, 1, soit une autre variable.
— Enfin on se contentera de 4 variables booléennes a, b, c et d.
On pourrait ainsi écrire un programme comme :

```
a := 1 ;
b := 1 ;
c := 1 ;
while(a) {
    if(c) {
       c := 0 ;
       a := b
    } else {
       b := 0 ;
       c := a
    }
}
```

Exercice 1 *Définir une hiérarchie de types OCaml permettant de représenter tous les programmes admis par la description ci-dessus.*

Pour éviter une analyse lexicale qui nous détournerait du cœur du sujet, on écrit les mots-clés du langage sur un seul caractère, on délimite le corps des if et des while par des accolades et on se dispense du mot-clé `else` (quitte à laisser un programme vide pour le second bloc du `if`).

Ainsi, notre programme exemple du début de l'énoncé s'écrit :

```
a:=1;
b:=1;
c:=1;
w(a){
  i(c){
    c:=0;
    a:=b
  }{
    b:=0;
    c:=a
  }
}
```

On a ici conservé les tabulations et les retours à la ligne pour que le programme reste lisible, mais on devrait même s'en dispenser également, et donc notre programme s'écrit finalement

```
a:=1;b:=1;c:=1;w(a){i(c){c:=0;a:=b}{b:=0;c:=a}}
```

Exercice 2 *Donner une grammaire décrivant ce langage.*

Exercice 3 *La grammaire que vous avez écrite est très probablement récursive gauche dans le cas de la séquence de programmes. Modifiez-la pour remédier à ce problème.*

2 Sémantique Opérationnelle Structurée (SOS)

La SOS est une méthode permettant de donner un sens à un programme dans un certain langage ou plus précisément donner une signification à chaque instruction, c'est-à-dire la manière dont elle s'exécute et ses effets.

L'exécution d'un programme est donnée par des transitions entre états notées $etat \longrightarrow etat\_suivant$, les états étant des couples notés $\Gamma \vdash (P)$, où Γ est un environnement et P est un programme exécuté dans cet environnement. Une transition est donc de la forme $\Gamma \vdash (P) \longrightarrow \Gamma' \vdash (P')$. Pour chaque forme possible de P, on indique que sera, après une étape d'exécution, le nouvel environnement Γ' et quelle est la suite du programme P'. On peut voir Γ et Γ' comme des états mémoire (partie *data*).

La première règle de transition s'applique au cas où P est une affectation :

$$\frac{}{\Gamma \vdash (i := expr) \longrightarrow \Gamma / i = [\![expr]\!]_\Gamma \vdash (\texttt{Skip})}$$

Ici, $[\![expr]\!]_\Gamma$ représente la valeur renvoyée par l'évaluation de *expr* dans l'environnement Γ, et la notation $\Gamma / i = v$ représente l'environnement obtenu à partir de Γ dans lequel toutes les associations sont conservées, sauf pour la variable i qui est associée à la valeur v. Une fois la modification faite dans l'environnement, il ne reste plus rien à exécuter et le programme devient donc `Skip`.

Les règles suivantes s'appliquent aux programmes qui sont des séquences, de la forme $P;Q$. Il faut d'abord exécuter le premier pas de P, puis considérer deux cas, suivant que la continuation de P est vide ou non. S'il ne reste rien à faire dans P après en avoir exécuté un pas, le programme devient le second membre de la séquence (règle de gauche). Sinon la règle de droite s'applique.

$$\frac{\Gamma \vdash (P) \quad \longrightarrow \quad \Gamma' \vdash (\texttt{Skip})}{\Gamma \vdash (P;Q) \quad \longrightarrow \quad \Gamma' \vdash (Q)} \qquad \frac{\Gamma \vdash (P) \quad \longrightarrow \quad \Gamma' \vdash (P') \qquad \text{avec } P' \neq \texttt{Skip}}{\Gamma \vdash (P;Q) \quad \longrightarrow \quad \Gamma' \vdash (P';Q)}$$

Enfin, cette dernière règle s'applique aux programmes qui sont des boucles conditionnelles : on « déplie » une itération de la boucle pour la remplacer par une condition, qui permettra de décider si on exécute le corps et recommence, ou s'il ne reste rien à faire.

$$\frac{}{\Gamma \vdash (\texttt{while } \textit{expr } P) \quad \longrightarrow \quad \Gamma \vdash (\texttt{if } \textit{expr } \texttt{then } (P ; \texttt{while } \textit{expr } P) \texttt{ else Skip})}$$

Exercice 4 *Écrire la (ou plutôt les) règles de transition de l'instruction* `if`*.*

Exercice 5 *Étendre votre grammaire et vos types pour traiter les affectations de la forme* `V:=#` *où* `V` *est une variable, et qui signifiera « remplacer la valeur de la variable par sa négation ».*

Exercice 6 (facultatif) *Améliorer votre grammaire et vos types pour pouvoir stocker des entiers dans les variables, et écrire des expressions arithmétiques simples dans les membres droits des affectations, ainsi que dans les conditions des if et des while.*

3 Implémentation de l'analyseur

Exercice 7 *Implémenter un analyseur syntaxique en OCaml pour la grammaire obtenue à la fin du TD.*

Exercice 8 *Écrire quelques programmes While pour tester votre analyseur. (Vous pouvez augmenter le nombre de variables disponibles si vous en ressentez le besoin.)*

Exercice 9 (facultatif) *Améliorer l'analyseur pour qu'il accepte des programmes avec des blancs arbitraires : espaces, indentations et retours à la ligne.*

4 Mécanique d'environnement et interpréteur

Le principe d'un interpréteur est d'exécuter pas à pas les instructions d'un programme dans un certain langage. Il s'agit ici d'une donnée arborescente Ocaml, il faut donc parcourir l'arbre représentant le programme en exécutant au fur et à mesure les instructions rencontrées ce qui revient à traduire les règles de transition en Ocaml.

Il nous faudra pour cela une modélisation de l'environnement contenant les valeurs courantes des variables.

On représente l'environnement par un `bool array` de taille 4, dans lequel la cellule d'indice 0 contient la valeur de a, celle d'indice 1 la valeur de b, et ainsi de suite.

Exercice 10 *Écrire des fonctions permettant respectivement :*

- *d'initialiser cet environnement (avec toutes les variables à 0) ;*
- *de lire la valeur d'une variable ;*
- *de modifier la valeur d'une variable ;*
- *d'exécuter une instruction d'affectation.*

Exercice 11 *Écrire une fonction* `faire_un_pas_p` *qui rend le nouveau programme après exécution d'une transition, ainsi qu'une fonction* `faire_un_pas_e` *qui rend le nouvel environnement après exécution d'une transition.*

```
val faire_un_pas_p : programme → environnement → programme
val faire_un_pas_e : programme → environnement → environnement
```

Un programme est terminé lorsqu'il est égal à `Skip`.

Exercice 12 *Écrire une fonction* `executer` *qui exécute un programme jusqu'à ce qu'il soit terminé.*

```
val executer : programme → environnement
```

Indication. Attention, si le programme à exécuter boucle l'interpréteur bouclera en l'exécutant. D'autre part, on pourra utiliser une fonction auxiliaire comportant un environnement comme argument supplémentaire et qui sera appelée avec l'environnement initial.

Exercice 13 (facultatif) *Instrumenter votre interpréteur pour qu'il compte et affiche le nombre de pas nécessaires pour interpréter le programme.*

Exercice 14 (facultatif) *Implémenter la sémantique natuelle sur le même principe et vérifier que les deux sémantiques correspondent sur vos programmes de test.*