Analyse syntaxique et Sémantique Opérationnelle Structurée

25 novembre 2020

1 Préliminaires théoriques

1.1 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.*

1.2 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 configurations notées *config* $\longrightarrow$ *config_suivante*, les configurations ayant deux formes possibles :

— un couple noté $E,\ (P)$, où E est un état et P est un programme à exécuter dans cet état ;

— un état simple sans programme restant à exécuter, ce que l'on notera $E,\ ()$.

Une transition est donc de la forme $E,\ (P) \quad \longrightarrow \quad E',\ (P')$, les programmes P et P' étant éventuellement vides. Pour chaque forme possible de P, on indique que sera, après une étape d'exécution, le nouvel état E' et quelle est la suite du programme P'. On peut voir E et E' comme des états mémoire (partie *data*).

La première règle de transition s'applique au cas où P est l'instruction vide :

$$\frac{}{E, (\texttt{Skip}) \quad \longrightarrow \quad E\ , ()}$$

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

$$\frac{}{E, (i := expr) \quad \longrightarrow \quad E\ /\ i = [\![expr]\!]_E\ , ()}$$

Ici, $[\![expr]\!]_E$ représente la valeur renvoyée par l'évaluation de *expr* dans l'état E, et la notation $E \,/\, i = v$ représente l'état obtenu à partir de E 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'état, il ne reste plus rien à exécuter.

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.

$$\frac{E,\ (P) \quad \longrightarrow \quad E',\ ()}{E,\ (P;Q) \quad \longrightarrow \quad E',\ (Q)} \qquad \frac{E,\ (P) \quad \longrightarrow \quad E',\ (P') \qquad \text{avec } (P') \neq ()}{E,\ (P;Q) \quad \longrightarrow \quad E',\ (P';Q)}$$

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.

$$\frac{}{E, (\texttt{while}\ expr\ P) \quad \longrightarrow \quad E,\ (\texttt{if}\ expr\ \texttt{then}\ (P\,;\ \texttt{while}\ expr\ P)\ \texttt{else}\ \texttt{Skip})}$$

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 ».*

2 Partie principale

2.1 Implémentation de l'analyseur simple

Exercice 6 *Implémenter un analyseur syntaxique en OCaml pour la grammaire obtenue. Une meilleure note sera accordée si vous utilisez et comprenez la technique des combinateurs d'analyseurs ; sinon, procéder avec des* `let ... in`.

Exercice 7 *Écrire quelques programmes While pour tester votre analyseur. (Vous pouvez augmenter le nombre de variables disponibles si vous en ressentez le besoin.)*

Exercice 8 (facultatif) *Améliorer l'analyseur pour qu'il accepte des programmes avec des blancs arbitraires : espaces, indentations et retours à la ligne.*

2.2 Mécanique d'état 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 faudra pour cela une modélisation de l'état contenant les valeurs courantes des variables. La représentation de l'état est laissée libre. Il est recommandé d'utiliser le système de modules de OCaml pour s'abstraire de l'implémentation choisie. Attention : le choix d'une structure de donnée mutable (basée sur le type `array` par exemple) est tentant mais comporte des dangers qu'il convient d'identifier soigneusement.

Exercice 9 *Écrire des fonctions permettant respectivement :*
- *d'initialiser cet état (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 10 *Traduire en OCaml le type* `config` *comprenant deux constructeurs comme vu en LT, mais utilisant la structure de donnée que vous avez choisie pour les états.*
Écrire une fonction `faire_un_pas` *qui rend la nouvelle configuration après exécution d'une transition.*

```
val faire_un_pas : programme → état → config
```

L'exécution d'un programme est terminée dans une configuration sans programme restant à exécuter.

Exercice 11 *Écrire une fonction* `executer` *qui exécute un programme jusqu'à ce qu'il soit terminé.*

```
val executer : programme → état
```

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 état comme argument supplémentaire et qui sera appelée avec l'état initial.

2.3 Preuves Coq

Les questions indiquées ici sont issues de l'énoncé du TD 9 de LT : elles en proviennent directement, ou du moins supposent données les définitions posées dans ce fichier, telles que `evalA`, `evalB`, `config`, `SOS_1` et `SOS`. Se reporter au fichier `TD09_SOS_While.v` pour plus de détails.

Exercice 12 *Propriétés générales. Il est possible de commencer l'exercice suivant sans avoir fait celui-ci.*
- *Formaliser la propriété indiquant que la relation SOS est transitive.*
- *Démontrer cette propriété. (Sinon on pourra l'utiliser tout de même).*
- *Énoncer en français la propriété indiquée par le théorème* `SOS_seq`*. On pourra au besoin ce théorème sans le démontrer.*

Exercice 13 *Programme WHILE* `Pcarre_2`*.*
- *Démontrer que* `Pcarre_2` *mène d'un état avec $i = 0$, $x = 0$ et $y = 1$ à une configuration intermédiaire où* `Pcarre_2` *peut être exécuté en partant d'un état avec $i = 1$, $x = 1$ et $y = 3$.*
- *Démontrer le lemme* `SOS_Pcarre_inf_1er_tour` *et indiquer ce que signifie son énoncé.*
- *Indiquer la signification du théorème* `SOS_Pcarre_2_2e_tour`*.*
- *Indiquer la signification du théorème* `SOS_Pcarre_2_fini` *et le démontrer.*
- *Démontrer que* `Pcarre_2` *mène d'un état avec $i = 0$, $x = 0$ et $y = 1$ à une configuration finale d'état $i = 2$, $x = 4$ et $y = 5$. On pourra utiliser la transitivité de SOS.*

Exercice 14 *Programmes WHILE* `Pcarre` *et* `Pcarre_inf`*.*
- *Indiquer la signification de* `SOS_corps_carre`*. Démontrer ce théorème.*
- *Indiquer la signification de* `SOS_corps_carre_inter`*. Démontrer ce théorème.*
- *Indiquer la signification de* `SOS_Pcarre_tour`*. Démontrer ce théorème.*

— *Indiquer la signification de* `SOS_Pcarre_n_fini`. *Démontrer ce théorème.*
— *Expliquer en français la démonstration de* `SOS_Pcarre_2_fin_V2`.
— *Indiquer la signification de* `SOS_Pcarre_inf_tour`. *Démontrer ce théorème.*
— *Indiquer la signification de* `SOS_Pcarre_inf_n`. *Démontrer ce théorème.*

Exercice 15 *Version fonctionnelle de* `SOS_1`. *Cela permet notamment d'éviter de remplacer de nombreux* `eapply` *des exercices précédents par un* `apply ... with c` *où la configuration c est obtenue par cette fonction ; autrement dit, avancer dans la preuve en connaissance de cause.*

— *Définir une version fonctionnelle de* `SOS_1`.
— *Utiliser cette fonction comme oracle dans une des preuves effectuées auparavant, par exemple la première de l'exercice 13. Le but est de remplacer* `eapply SOS_again` *par* `apply SOS_again with c` *où c est une configuration intermédiaire que l'on peut calculer. Il est recommandé de bien s'organiser, en nommant à l'avance les programmes « restant à exécuter ».*

3 Extensions facultatives

Différentes extensions sont proposées, chaque équipe peut en choisir une ou plusieurs et les approfondir à sa guise. Vous pouvez également imaginer d'autres extensions, si besoin consulter les enseignants le cas échéant.

3.1 Interpréteur

— Instrumenter votre interpréteur de SOS pour qu'il compte et affiche le nombre de pas nécessaires pour interpréter le programme.
— Proposer un mode interactif pas à pas.

3.2 Analyse lexicale et syntaxique

On cherche ici à analyser un texte utilisant des conventions correspondant aux usages des langages de programmation : identificateurs de plus d'un caractère, entiers de plusieurs chiffres (un chiffre c est un caractère entre `'0'` et `'9'`, et sa valeur entière est `int_of_char` c $-$ `int_of_char` $'0'$), expressions arithmétiques et booléennes,...

Il conviendra de procéder de manière incrémentale, en partant d'une version très simple et en l'enrichissant progressivement. Par exemple pour le type `token` ci-dessous, commencer par une version de base ne comportant que très peu de cas.

— Identifier les différents lèxèmes (y compris les symboles spéciaux) et concevoir un type `token` pour eux.
— Réaliser un analyseur lexical qui prend une séquence de caractères et rend le premier token en tête de cette séquence.
  Pour les entiers, utiliser le schéma de Horner (après lecture d'un premier chiffre, passer l'entier correspondant comme accumulateur et à chaque nouveau chiffre, prendre pour nouvel accumulateur l'accumulateur précédent fois 10 additionné à la valeur entière du nouveau chiffre ; ceci peut se faire à mesure de l'analyse lexicale, ou a posteriori sur une liste de chiffres).
— Compléter cet analyseur en supprimant les espaces et caractères de retour à la ligne précédant un véritable lexème.

— Écrire un programme qui itère sur le précédent de façon à produire la liste de lexèmes correspondant à une séquence de caractères.
— Écrire un programme qui enchaîne une analyse lexicale avec une analyse syntaxique.
— Enrichir la grammaire de l'analyseur syntaxique pour accepter des expressions comprenant des opérateurs binaires ; attention à bien concevoir la grammaire pour éviter les récursions à gauche. Voici un modèle de grammaire pour les expressions additives :

```
E ::= T E'
E' ::= + T E' | ε
T  ::= atome | ( E )
```

On pourra la compléter en prenant pour `T` des expressions multiplicatives au lieu d'atomes, ce qui permet de gérer les priorités entre + et *. Un atome est un entier ou un identificateur.
Attention : l'AST de 3 + 2 + 1 correspond à (3 + 2) + 1, ce qui se voit mieux avec des soustractions : 3 - 2 - 1. L'AST se construit donc dans le même esprit que le schéma de Horner, par passage d'accumulateur.
— On pourra gérer les expressions arithmétiques et booléennes soit par des types et des grammaires séparées, à l'instar de `Aexp` et `Bexp` en LT, soit ensemble.

3.3 Changement de représentation des séquences

Utiliser des listes paresseuses au lieu des listes. A faire avec des changements minimaux : ne reprogrammer que les primitives comme `terminal`, `epsilon`, `return`, `+>` et `++>`.

3.4 Sémantique naturelle

Implémenter la sémantique naturelle sur le même principe que la SOS et vérifier que les deux sémantiques correspondent sur vos programmes de test.

3.5 Ajout de threads

Ajouter au niveau des instructions un opérateur de composition parallèle. Concevoir les règles de SOS associées.

Au nivau de l'interpréteur, prendre en compte le fait que plusieurs exécutions sont possibles. Plusieurs politiques sont possibles...

3.6 Preuves Coq

Exercice 16
— *Démontrer le théorème* `SOS_seq`.

Exercice 17 *Version fonctionnelle de* `SOS_1` *(appelée* `f_SOS_1`*)*
— *Démontrer que* `f_SOS_1` *est correcte : elle rend une configuration d'arrivée de* `SOS_1`.
— *Démontrer que* `f_SOS_1` *est complète : toute configuration d'arrivée de* `SOS_1` *est obtenue par* `f_SOS_1`.

Exercice 18 *Programme* `Pcarre` *(fin).*
Énoncer et démontrer le théorème général attendu pour Pcarre.

Exercice 19 *SN et SOS.*

Énoncer et démontrer des théorèmes convenables reliant les sémantiques SN et SOS du langage WHILE.

Exercice 20 *Programmes infinis et finis.*

Généraliser l'observation effectuée à propos des théorèmes sur `Pcarre` *et* `Pcarre_inf`. *Énoncer et démontrer un théorème valable sur une classe de programmes WHILE dont ces programmes font partie.*