Analyse syntaxique et sémantique

27 novembre 2022

L'énoncé comporte des parties obligatoires et de nombreuses parties facultatives. Des instructions plus précises seront données séparément sur leur répartition et le travail attendu. Les questions plus difficiles sont indiquées par des étoiles plus ou moins nombreuses.

1 Préliminaires théoriques

1.1 Définition et analyse d'un langage de programmation simple

On considère une variante appelée WHILEb du langage WHILE vu en LT, où 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 (constituée d'une expression et d'un corps ; la condition d'arrêt étant que l'expression vaut 0).

Dans un premier temps, on considérera un version simplifiée de WHILEb nommée WHILEb^{--} où :
- 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 WHILEb^{--} comme :

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

Exercice 1.1.1 *Définir une hiérarchie de types OCaml permettant de représenter tous les programmes admis par la description ci-dessus.*

Dans un premier temps on va économiser la phase traditionnelle d'analyse lexicale. Pour cela 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 si on s'en dispense, notre programme s'écrit :

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

Exercice 1.1.2 *Donner une grammaire décrivant le langage* WHILEb^{--}.

Exercice 1.1.3 *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.*

Exercice 1.1.4 *(*) Cet exercice peut se traiter après avoir réalisé les questions de la partie principale sur le langage de base* WHILEb^{--}.

Le langage WHILEb *est obtenu à partir de* WHILEb^{--} *en étendant le langage d'expressions de façon à accepter, les constantes 0 ou 1 et les variables booléennes a, b, c ou d, ainsi que, par ordre de priorité croissante :*

- *des disjonctions booléennes écrites avec l'opérateur infixe* `'+'`*, associant à gauche,*
- *des conjonctions booléennes écrites avec l'opérateur infixe* `'.'` *associant à gauche,*
- *des négations booléennes écrites avec l'opérateur préfixe* `'!'`
- *des expressions parenthésées*

Exemple : `a+b.0+!1`*, qui se lit de la même façon que* `a+(b.0)+(!1)` *et que* `(a+(b.0))+(!1)`.

Pour cela on se base sur la grammaire suivante :

```
C ::= '0' | '1'
V ::= 'a' | 'b' |  'c' |  'd'
A ::= C | V
```

```
E ::= E '+' T   |   T
T ::= T '.' F   |   F
F ::= '!' F  |  A  |  '(' E ')'
```

Donner une grammaire non récursive à gauche de ce langage d'expressions.

1.2 Sémantique naturelle (SN), dite aussi sémantique opérationnelle à grands pas

Rappels de cours de LT. La SN est une méthode permettant de donner un sens à un programme dans un certain langage ou plus précisément donner une signification à chaque programme, c'est-à-dire la manière dont il s'exécute et ses effets. L'exécution d'un programme est donnée par des transitions entre états notées

$$\textit{état initial} \xrightarrow{\textit{programme}} \textit{état final}$$

signifiant que l'exécution du programme représenté par *programme* à partir de *état initial* conduit à *état final*.

Les transitions possibles sont exprimées inductivement par des règles. Les règles pour le programme `Skip` et pour une affectation i`:=`$expr$ n'ont pas de prémisse.

$$\frac{}{s \xrightarrow{\texttt{Skip}} s} \qquad\qquad \frac{}{s \xrightarrow{v\texttt{:=}expr} \textit{update}\ s\ v\ [\![expr]\!]_s}$$

La seconde règle est exprimée avec une fonction *update* qui rend l'état s dans lequel la variable v a pris la valeur de *expr* évaluée dans l'état s, écrite $[\![expr]\!]_s$.

La règle pour la séquence P ; Q (le programme P suivi du programme Q) comporte deux prémisses, une indiquant que l'exécution de P à partir de s_1 aboutit à s_2 et une indiquant que l'exécution de Q à partir de s_2 aboutit à s_3 ; l'exécution de P ; Q à partir de s_1 aboutit alors à s_3.

$$\frac{s_1 \xrightarrow{P} s_2 \qquad s_2 \xrightarrow{Q} s_3}{s_1 \xrightarrow{P\ ;\ Q} s_3}$$

Pour un programme de la forme `while` *expr* P, on a deux règles selon que l'évaluation de *expr* dans l'état s_1 donne *false* ou *true*.

$$\frac{[\![expr]\!]_{s_1} = \textit{false}}{s_1 \xrightarrow{\texttt{while}\ expr\ P} s_1} \qquad\qquad \frac{[\![expr]\!]_{s_1} = \textit{true} \qquad s_1 \xrightarrow{P} s_2 \qquad s_2 \xrightarrow{\texttt{while}\ expr\ P} s_3}{s_1 \xrightarrow{\texttt{while}\ expr\ P} s_3}$$

Exercice 1.2.1 *Écrire les règles de SN pour un programme de la forme* `if` *expr* `then` P `else` Q.

2 Partie principale

Chaque équipe devra choisir entre la partie 2.3 (option 1 sur LT/SN) et la partie 2.4 (option 2 sur LT/SOS). L'option 2.3 (bien réalisée) suffira pour donner une note correcte. L'option 2.4, plus complexe et concernant la fin du cours de LT, donnera bien plus de points. À titre indicatif : il sera plus difficile d'atteindre 14 ou delà avec l'option 2.3 qu'avec l'option 2.4.

2.1 Implémentation de l'analyseur simple

Exercice 2.1.1 *Implémenter un analyseur syntaxique en OCaml pour la grammaire obtenue du langage* WHILEb^{--}. *Une meilleure note sera accordée si vous utilisez et comprenez la technique des combinateurs d'analyseurs ; sinon, procéder avec des* `let ... in`.

Exercice 2.1.2 *Écrire quelques programmes* WHILEb^{--} *pour tester votre analyseur. (Vous pouvez augmenter le nombre de variables disponibles si vous en ressentez le besoin.)*

Exercice 2.1.3 *(*) Compléter l'analyseur pour accepter le langage d'expressions considéré à la fin de la partie 1.1.*

Exercice 2.1.4 (facultatif) *(*) Améliorer l'analyseur pour qu'il accepte des programmes avec des blancs arbitraires : espaces, indentations et retours à la ligne.*

2.2 Exécution d'un programme WHILEb

Rappels de LT : il est possible de lire les règles de SN comme une fonction prenant en entrée un programme, un état initial et retournant l'état final spécifié. On a vu en cours qu'on ne peut pas écrire une telle fonction en Coq avec un `Fixpoint` élémentaire. Cela est dû à une restriction spécifique aux besoins d'un assistant de preuve, restriction qui n'a pas de raison d'être dans les langages de programmation usuels.

Exercice 2.2.1 *Choisir un type de donnée OCaml pour représenter l'état d'un programme* WHILEb^{--} *(on pourra s'insiprer de celui sur lequel on a travaillé en LT). Écrire un programme OCaml qui, étant donné un AST* WHILEb *simplifié et un état initial, rend l'état final obtenu après exécution tel que spécifié par la sémantique SN.*

Pour tester ce programme, on enchaînera la production d'un AST par l'analyse syntaxique (partie 2.1) et son exécution selon la SN.

Exercice 2.2.2 *(*) Compléter l'exercice précédent pour prendre en entrée le langage* WHILEb.

2.3 Preuves sur la SN (option 1)

Il s'agit ici de rendre votre solution du TD6 de LT, consacré à la SN du langage WHILE. À titre indicatif, on portera une attention plus particulière. aux exercices ci-dessous.

IMPORTANT : pour les élèves qui ont bien avancé et choisissent l'option 2 (voir l'introduction de la partie 2), allez directement à la partie suivante (SOS).

Exercice 2.3.1 *Programme WHILE* `Pcarre_2`.
Montrer le lemme `reduction_Pcarre_2` *du TD 6.*

Exercice 2.3.2 *De SN vers SN'.*
Montrer le lemme `SN_SN'` *du TD 6.*

Exercice 2.3.3 *(Facultatif **) De SN' vers SN.*
Montrer le lemme `SN'_SN` *du TD 6.*

2.4 Preuves sur la SOS (option 2)

Les questions indiquées ici sont issues des énoncés du TD 7 de LT : elles en proviennent directement, ou du moins supposent données les définitions posées dans ces fichiers, telles que `evalA`, `evalB`, `config`, `SOS_1` et `SOS`. Se reporter au fichier `TD07_SOS_winstr.v` pour plus de détails.

Exercice 2.4.1 *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 utiliser ce théorème sans le démontrer.*

Exercice 2.4.2 *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 2.4.3 *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 2.4.4 *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 2.4.2. 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 (**) 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.
— 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.

3.2 (*) Changement de représentation des séquences

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

3.3 (*) 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 3.3.1 *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 3.3.2 *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 3.3.3 *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.

3.4 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.

3.5 (** – ***) Ajout de threads

Ajouter au niveau des instructions un opérateur de composition parallèle noté `//`. On pourra par exemple accepter le programme suivant :

```
a := 1 ;
b := 0 ;
{ b := a ; c := b //
   c := 0 ; a := c }
```

L'exécution des deux séquences `b := a ; c:= b` et `c := 0 ; a := c` s'effectue avec un entrelacement quelconque des affectations que l'on supposera atomiques (ininterruptibles), ce qui donne plusieurs résultats possibles.

(**) Concevoir les règles de SOS associées.

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

(***) Concevoir les règles de SN associées. Observer que certains comportements possibles avec SOS ne peuvent être obtenus avec SN.

3.6 (****) Autre approche de l'analyse

D'autres bibliothèques de combinateurs de parsers ont été proposées, par exemple celle-ci : https://discuss.ocaml.org/t/ann-reparse-2-0-0/6868. L'expérimenter sur le langage proposé dans ce sujet. Comparer cette approche avec les combinateurs introduits en PF tels que `++>` et `epsilon_res`, selon plusieurs critères :

— le confort d'utilisation au niveau de l'écriture de programmes ;
— la facilité à comprendre les mécanismes sous-jacents ;
— l'efficacité des programmes d'analyse obtenus (on demande ici une réflexion intuitive, pas forcément la mise en œuvre d'un banc de mesure).

3.7 Preuves Coq

Exercice 3.7.1

— *(** – ***) Démontrer le théorème* `SOS_seq`. *(Court mais non trivial).*

Exercice 3.7.2 *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 3.7.3 *(** – ***) Programme* `Pcarre` *(fin).*

Énoncer et démontrer le théorème général attendu pour Pcarre.

Exercice 3.7.4 *(***) SN et SOS.*

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

Exercice 3.7.5 *(****) 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.*

3.8 (**) Compilation d'expressions arithmétiques

Exercices du TD11 de LT.

3.9 (****) Vers un compilateur de WHILE

Sujet expérimental. Peut s'effectuer en OCaml ou en Coq, sur la base de la dernière séquence CM-TD.