De l'informel au formel

Amélie LEDEIN

D'après le matériel de cours de Catherine DUBOIS,
Guillaume BUREL, et tant d'autres !

Ce document constitue l'autre partie du projet, concernant l'UE Projet Logique : cette partie orientée "sémantique" demande, dans un premier temps, de travailler avec Coq, puis, dans un second temps, avec $\mathbb{K}$.

L'objectif de ce projet est d'aborder la sémantique formelle grâce à Coq, puis $\mathbb{K}$.

1 Expressions arithmétiques

1. Formaliser, en Coq, la sémantique opérationnelle à grands pas du langage des expressions arithmétiques.

   Une expression est soit une constante entière, une variable, ou une expression de la forme $e_1 + e_2$, $e_1 * e_2$, $-e$.

   Les valeurs sémantiques sont des entiers relatifs.

2. Écrire un interprète pour le langage `AExp`.

3. Démontrer que le langage `AExp` est déterministe.

2 Expressions booléennes

1. Formaliser, en Coq, la sémantique opérationnelle à grands pas du langage des expressions booléennes.

   Une expression est soit une constante entière, une variable, ou une expression de la forme $e_1 = e_2$, *not* e, $e_1 \wedge e_2$.

   Les valeurs sémantiques sont des entiers relatifs.

2. Écrire un interprète pour le langage `BExp`.

3. Démontrer que le langage `BExp` est déterministe.

3 IMP : un langage impératif

1. Formaliser la sémantique opérationnelle à grands pas de `IMP`.

   Vous ajouterez notamment les commandes que sont l'affectation, la séquence, la construction `if-then-else` et la boucle `repeat`.

2. Écrire un interprète pour le langage `IMP`.

3. Montrer que ce langage est déterministe.

4. Définir le prédicat `equivalent_com` spécifiant l'équivalence de deux commandes.

5. Montrer que $\mathsf{while}\ b\ \mathsf{do}\ c \equiv \mathsf{if}\ b\ \mathsf{then}\ c; \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}$. ($\equiv$ symbole de l'équivalence de deux commandes)

6. On souhaite étendre le langage des instructions afin de permettre l'écriture d'instructions de la forme : $\mathsf{repeat}\ c\ \mathsf{until}\ b$ où $c \in E_C$ et $b \in E_B$. La sémantique informelle de cette instruction est : "exécuter c jusqu'à ce que b prenne la valeur true".

   Modifier la formalisation précédente de manière à spécifier cette nouvelle construction.

7. Introduire une construction d'affectation multiple de la forme $x, y := e_1, e_2$ qui réalise simultanément les deux affectations. Les expressions e_1 et e_2 sont évaluées avec la même valuation. Avec cette construction, l'échange des valeurs de deux variables x et y s'écrit simplement $x, y := y, x$.

8. Montrer sous quelles conditions $x, y := e_1, e_2$ est équivalent à la séquence $x := e_1; y := e_2$. On pourra poser les axiomes nécessaires à la preuve de la propriété que vous aurez établie.

9. **Ajout des variables locales.**
On ajoute à présent une construction permettant la présence de variables locales dans les expressions. Par exemple, on veut pouvoir considérer les expressions de la forme $\mathsf{let}(x, 3 + z, x + y)$. Dans cette expression, les valeurs de z et y sont obtenues à partir de la valuation utilisée pour évaluer l'expression, tandis que x est une variable locale (re)définie lors de l'évaluation. On dit qu'une telle variable est liée, tandis que z et y sont libres. On ajoute donc au système d'inférence la règle :

$$(8)\frac{a_1 \quad a_2}{\mathsf{let}(x, a_1, a_2)}$$

permettant de considérer des expressions de la forme $\mathsf{let}(x, a_1, a_2)$, où $x \in V$.

La règle d'inférence définissant la sémantique opérationnelle de la construction let est donnée ci-dessous :

$$(A_{let})\frac{\langle a_1, \sigma\rangle \rightsquigarrow n_1 \quad \langle a_2, \sigma[x \leftarrow n]\rangle \rightsquigarrow n_2}{\langle \mathsf{let}(x, a_1, a_2), \sigma\rangle \rightsquigarrow n_2} \ (n_1, n_2 \in \mathbb{Z})$$

$$\text{avec } \sigma[x \leftarrow n](y) = \begin{cases} n & \text{si } y = x \\ \sigma(y) & \text{sinon} \end{cases}$$

Ajouter cette construction à l'interpréteur et à la formalisation Coq (prédicat d'évaluation à grands pas et preuve que le langage est déterministe.

4 MiniML : un langage fonctionnel

- Formaliser, en Coq, la sémantique opérationnelle à grands pas de MiniML.

  Une expression est soit une constante entière, une variable, une addition de deux expressions, une lambda-abstraction de la forme (`fun x. e`) ou une application (le langage est donc un lambda-calcul étendu avec des constantes entières).

  Les valeurs sémantiques sont des entiers relatifs ou des fermetures. Une fermeture est un couple formé d'une abstraction et d'un environnement (valuation). Le prédicat d'évaluation est noté `<e,s> --> v` et se lit la valeur de `e` relativement au contexte `s` est `v`.

- Écrire un interprète pour ce langage fonctionnel.

- Démontrer que le langage est déterministe.

- Quelle est la valeur de ((fun x. x)(x + 1)) + x relativement à la valuation s telle que s(x)=0 ? Vérifiez votre réponse avec votre interprète. Justifiez formellement votre réponse avec Coq (démontrez le lemme correspondant).

- On ajoute au langage précédent la construction let x = e1 in e2 où x est une variable, e1 et e2 des expressions. Informellement évaluer une telle expression consiste à évaluer e1, soit v sa valeur, évaluer e2 dans un environnement où x est associée à la valeur v. La valeur obtenue est alors la valeur de l'expression complète let x = e1 in e2.

  Modifier le langage précédent pour incorporer cette nouvelle construction. Reprendre la preuve et l'interprète.

- Quelle est la valeur de (let x = x+1 in x)+x relativement à la valuation s telle que s(x)=0 ? Vérifiez votre réponse avec votre interprète. Justifiez formellement votre réponse avec Coq.

- Démontrer en Coq que les deux expressions let x = e1 in e2 et (fun x. e2) e1 sont équivalentes.

- Définir en Coq le prédicat `est_bien_typé`. Les règles de typage sont celles du lambda-calcul simplement typé.