TP noté Sémantique

1 Expressions

On commence par définir un langage, `Exp`, d'expressions arithmétiques booléennes. Ainsi une expression est soit une constante entière, une variable, ou une expression de la forme $e_1 + e_2$, $e_1 * e_2$, $e_1 = e_2$, *not* e, $e_1 \wedge e_2$.

La syntaxe abstraite et la sémantique opérationnelle à grands pas de ce langage sont définies formellement en Coq dans le fichier exp.v. Les valeurs sémantiques sont des entiers relatifs. Nous avons en effet une sémantique *à la C* pour les expressions booléennes. (false = 0, true = un entier différent de 0).

1. Ecrire en Ocaml un interprète pour le langage Exp.
2. Le fichier exp.v contient la formalisation Coq de la sémantique opérationnelle à grands pas du langage des expressions. Il contient également la preuve que le langage est déterministe. Cette preuve est partielle, `complétez la`.

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 définissant Exp 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 la formalisation Ocaml et à la formalisation Coq (prédicat d'évaluation à grands pas et preuve que le langage est déterministe.

2 Constructions impératives

1. Le fichier com.v contient la formalisation de la sémantique opérationnelle big step du petit langage impératif étudié en cours. Montrer que ce langage est déterministe.

2. Définir le prédicat `equivalent_com` spécifiant l'équivalence de deux commandes (de type `com -> com -> Prop`.

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

4. On souhaite étendre le langage des instructions afin de permettre l'écriture d'instructions de la forme : repeat c 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.

5. 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 $x, y := y, x$.

6. 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.