ISTIC
UFR INFORMATIQUE - ELECTRONIQUE

Licence STS 2ème année mentions Informatique & Mathématiques

Contrôle continu n°1 - jeudi 24 mars 2022 - durée : 1h30

Exercice 1 (Dénombrabilité) :

Question 1.1 : Soient E et F deux ensembles dénombrables. Montrez que le produit cartésien $E \times F$ est dénombrable.

Si E est dénombrable, il existe une bijection f de E dans $\mathbb{N}$.

Si F est dénombrable, il existe une bijection g de F dans $\mathbb{N}$.

Par composition de f et g, on peut définir la bijection h suivante :

$$\begin{array}{rccc} h: & E \times F & \rightarrow & \mathbb{N} \\ & (x,y) & \mapsto & g(f(x), f(y)) \end{array}$$

$E \times F$ est en bijection avec $\mathbb{N}$, $E \times F$ est donc dénombrable.

Exercice 2 (Réduction calculatoire) :

Question 2.1 : Notons Carré le problème de déterminer, étant donné un programme, s'il existe une valeur d'entrée pour laquelle ce programme rend un résultat égal au carré de cette valeur. Donnez une spécification formelle du problème Carré.

La spécification du problème Carré est la suivante :

$$\begin{array}{rccc} \text{Carré} : & \mathbb{P} & \rightarrow & \mathbb{B} \\ & \mathtt{p} & \mapsto & \exists \mathtt{d} \in \mathbb{D} : \mathcal{SEM}(\mathtt{p})(\mathtt{d}) = \mathtt{d}^2 \end{array}$$

Question 2.2 : Montrez par réduction du problème de l'arrêt que le problème Carré est indécidable.

Supposons Carré décidable. Soit *square* un programme résolvant le problème Carré. On peut alors construire un programme *halt* résolvant le problème de l'arrêt comme suit :

> algo halt(p,v)
> Construction du programme suivant :
> p'(x) : $p(v)$; 4
> Renvoyer la valeur : square(p')
> fin

Il existe bien une valeur d'entrée, égale à 2, pour laquelle le programme p' renvoie le carré de cette valeur. Le programme *halt* rend la valeur vrai si et seulement si le programme *square* rend vrai. Et celui-ci rend vrai si et seulement si l'exécution de p sur v s'arrête.

Le problème de l'arrêt étant indécidable, ce programme *halt* ne peut pas exister. Le programme *square* n'existe donc pas et le problème Carré est indécidable.

Exercice 3 (Programmes, problèmes et fonctions) :

Question 3.1 : Quelle importance cela a-t-il que les programmes soient des textes finis ?

Le fait que les programmes soient des textes finis détermine que l'ensemble des programmes est dénombrable. Or l'ensemble des fonctions est non dénombrable. Cela nous a donc permis de justifier qu'il y a des fonctions qu'aucun programme ne peut résoudre.

Question 3.2 : Quelle différence fait-on entre une fonction et un problème dans le cours ?

Un problème est un cas particulier de fonction à résultat booléen.

Question 3.3 : La preuve de l'indécidabilité du problème de l'arrêt est-elle une preuve :
— par diagonalisation ?
— par l'absurde ?
— par réduction ?
Justifiez en quelques lignes votre réponse.

C'est une preuve par l'absurde. On a supposé que le problème de l'arrêt était décidable et qu'un programme, nommé *halt*, le résolvait. Cette supposition nous a conduit à une contradiction. Ce programme *halt* ne peut donc pas exister et le problème de l'arrêt est ainsi indécidable.

Question 3.4 : Soit la propriété de programme $\exists y.\, \forall x : \mathcal{SEM}(p)(x) > y$. Que décrit-elle ? Est-elle extensionnelle ? Justifiez votre réponse.

Cette propriété de programme décrit l'ensemble des programmes dont les résultats dépassent un minimum. Chaque programme p possède son propre minimum y. Elle est extensionnelle car deux programmes ayant la même sémantique renverront les mêmes résultats pour chacune des valeurs d'entrée. Ces résultats seront supérieurs au même minimum.

Exercice 4 (Syntaxe et sémantique) :

Nous introduisons une nouvelle commande dont la syntaxe est *swap* (V_1, V_2) et dont la sémantique informelle est d'intervertir les valeurs de deux variables quelconques V_1 et V_2.

Question 4.1 : Proposez une modification de la grammaire du langage WHILE afin d'intégrer cette nouvelle commande.

On ajoute dans la partie commande de la grammaire la règle suivante :

$$\text{commande} \quad \rightarrow \quad \texttt{swap(variable,variable)}$$

Question 4.2 : Donnez la sémantique formelle de la commande *swap*.

Il faut veiller à ce que les deux variables soient évaluées sur la même mémoire m. La sémantique formelle de cette commande *iterate* peut être la suivante :

$$\mathcal{SEM}_c : \text{commande} \times \textit{mémoire} \rightarrow \textit{mémoire}$$
$$\mathcal{SEM}_c(\textit{swap}\,(v_1, v_2), m) = m[v_1 \rightarrow m(v_2)][v_2 \rightarrow m(v_1)]$$

Question 4.3 : Etendre la définition du trait Command et la fonction interpreterCommand pour implémenter la sémantique de *swap* dans le programme *interpreter*.

```
sealed trait Command
⋮
case class Swap(v1 : Variable, v2 : Variable) extends Command
⋮
```

Dans la programmation Scala, on veillera aussi à ce que les évaluations des variables se fassent bien sur la même mémoire et on fera attention à ne pas composer des affectations qui modifieraient séquentiellement la mémoire.

On évitera donc d'écrire :

```
def interpreterCommand(c : Command, m : Memory) : Memory = {
⋮
case Swap(v1, v2) => assign(v2, interpreterExpr(v1, m),
                                    assign(v1, interpreterExpr(v2, m), m))

⋮
}
```

ce qui revient à simuler les affectations en séquence : v1 := v2 ; v2 := v1. On voit que l'ancienne valeur de v1 a été écrasée par la première affectation. Programmer *swap* en séquence demande d'introduire une $3^{ème}$ variable : v3 := v1 ; v1 := v2 ; v2 := v3.

L'idée est plutôt d'évaluer les variables en amont des affectations :

```
def interpreterCommand(c : Command, m : Memory) : Memory = {
⋮
case Swap(v1, v2) => {
        val val1 : Value = interpreterExpr(v2, m) ;
        val val2 : Value = interpreterExpr(v1, m) ;
        assign(v2, val1, assign(v1, val2, m)) }
⋮
}
```