

Preuves de programmes – TP n° 3

Définitions inductives en Coq

I) Un compilateur d'expressions arithmétiques en Coq

Exercice 1 – Expressions arithmétiques

On considère le type d'expressions arithmétiques défini en Coq par :

```
Inductive expr : Type :=
  | Num : nat → expr          (* Constante entière *)
  | Plus : expr → expr → expr (* Somme de deux expressions *)
  | Mult : expr → expr → expr (* Produit de deux expressions *)
```

1. Définir la fonction d'évaluation correspondante `eval : expr → nat`
2. Tester cette fonction à l'aide de la commande `Eval compute in ...`

Exercice 2 – La machine à pile

On considère une machine à pile dont le jeu d'instructions est donné par :

```
Inductive instr : Set :=
  | PUSH : nat → instr (* Pousser un nombre sur la pile *)
  | ADD : instr          (* Ajouter les deux nombres au sommet de la pile *)
  | MUL : instr.         (* Multiplier les deux nombres au sommet de la pile *)
```

Les notions de programme et de pile sont définis par :

```
Definition prog := list instr.
Definition stack := list nat.
```

On utilisera les listes (polymorphes) de Coq dont les définitions sont chargées à l'aide de la commande `Require Export List`.

1. Définir une fonction `exec_instr : instr → stack → stack` exécutant une instruction dans une pile donnée et retourne l'état de la pile après l'exécution de l'instruction. Quel choix d'implémentation faites-vous dans le cas où la pile ne contient pas assez d'éléments pour exécuter une instruction ? Critiquez ce choix.
2. Définir une fonction `exec_prog : prog → stack → stack` exécutant un programme dans une pile donnée et retourne l'état de la pile après l'exécution de l'instruction.
3. Montrez que la fonction d'exécution de programme est associative :

```
forall (p1 p2 : prog) (s : stack),
  exec_prog (p1 ++ p2) s = exec_prog p2 (exec_prog p1 s).
```

Exercice 3 – Le compilateur

On considère la fonction de compilation définie par :

```
Fixpoint compile (e : expr) : prog :=
  match e with
  | Num n      ⇒ PUSH n :: nil
  | Plus e1 e2 ⇒ compile e2 ++ compile e1 ++ (ADD :: nil)
  | Mult e1 e2 ⇒ compile e2 ++ compile e1 ++ (MUL :: nil)
  end.
```

1. Exprimez en Coq puis démontrez la correction de la fonction de compilation.
2. Quelle est la complexité de la fonction de compilation ?

Une technique de compilation standard¹ consiste à produire le code en sens inverse, à l'aide d'une fonction récursive qui prend en argument l'expression à compiler et le code qui suit :

```
Fixpoint compile_cont (e : expr) (p : prog) : prog :=
  match e with
  | Num n      ⇒ PUSH n :: p
  | Plus e1 e2 ⇒ compile_cont e2 (compile_cont e1 (ADD :: p))
  | Mult e1 e2 ⇒ compile_cont e2 (compile_cont e1 (MUL :: p))
  end.
```

Le programme p est parfois appelé une continuation. De cette fonction on déduit une fonction de compilation à la complexité linéaire :

```
Definition compile_opt (e : expr) := compile_cont e nil.
```

3. Montrez que `compile_opt` produit exactement le même code que `compile`. En déduire que cette nouvelle fonction de compilation est correcte.

¹C'est cette technique qui est utilisée dans l'implémentation courante du compilateur Caml. En pratique, cette façon de produire le code permet à chaque instant de connaître la suite du code, et d'utiliser cette connaissance pour effectuer des optimisations à la volée.

II) Définition inductives de prédicats

Exercice 4 – Définitions de la clôture réflexive-transitive

Sur un type de données $T : \text{Type}$ fixé, on cherche à définir la clôture réflexive-transitive d'une relation $R : T \rightarrow T \rightarrow \text{Prop}$, qui est par définition la plus petite relation réflexive et transitive contenant la relation R . Il est naturel d'introduire en Coq cette notion au moyen de la définition inductive suivante (paramétrée par la relation R) :

```
Inductive clos1 (R : T → T → Prop) : T → T → Prop :=
| cl1_base : forall x y, R x y → clos1 R x y
| cl1_refl : forall x, clos1 R x x
| cl1_trans : forall x y z, clos1 R x y → clos1 R y z → clos1 R x z.
```

Cependant, il est souvent commode dans les démonstrations de définir la notion de clôture réflexive-transitive d'une manière un peu différente, à savoir comme la relation R' engendrée par les règles suivantes :

1. Pour tout x , $R' x x$ (cas de base) ;
2. Si $R' x y$ et $R y z$, alors $R' x z$ (cas inductif).

Cette définition alternative se modélise en Coq au moyen de la définition inductive suivante :

```
Inductive clos2 (R : T → T → Prop) : T → T → Prop :=
| cl2_refl : forall x, clos2 R x x
| cl2_next : forall x y z, clos2 R x y → R y z → clos2 R x z.
```

Le but de cet exercice est de montrer l'équivalence des deux définitions. Pour ce faire, on pourra suivre les étapes suivantes :

1. Montrer que $\text{clos2 } R x y$ entraîne $\text{clos1 } R x y$ (pour tous $x, y : T$).
2. Montrer que $R x y$ entraîne $\text{clos2 } R x y$ (pour tous $x, y : T$).
3. Montrer que $\text{clos2 } R$ est une relation transitive.
4. En déduire que $\text{clos1 } R x y$ entraîne $\text{clos2 } R x y$ (pour tous $x, y : T$).
5. Montrer que l'opération de clôture réflexive-transitive est idempotente, c'est-à-dire que : $\text{clos1 } (\text{clos1 } R) x y \Leftrightarrow \text{clos1 } R x y$ pour tous $x, y : T$.

Exercice 5 – Définitions de la clôture réflexive-transitive (suite)

On travaille toujours avec un type $T : \text{Type}$ fixé.

1. Définir la relation identité $\text{id} : T \rightarrow T \rightarrow \text{Prop}$ ainsi que l'opérateur de composition de relations $\text{comp} : (T \rightarrow T \rightarrow \text{Prop}) \rightarrow (T \rightarrow T \rightarrow \text{Prop}) \rightarrow T \rightarrow T \rightarrow \text{Prop}$.
2. Définir une fonction $\text{puiss} : (T \rightarrow T \rightarrow \text{Prop}) \rightarrow \text{nat} \rightarrow T \rightarrow T \rightarrow \text{Prop}$ telle que $\text{puiss } R n$ calcule la puissance n -ième de la relation (par l'opération de composition).

Une troisième définition de la clôture réflexive-transitive d'une relation R est donnée par la réunion des puissances successives de R , c'est-à-dire par :

```
Definition clos3 (R : T → T → Prop) (x y : T) := exists n, puiss R n x y.
```

3. Montrer que cette définition est équivalente aux deux précédentes (clos1 et clos2).