Un compilateur d'expressions arithmétiques en Coq

TP 3

■ ■ ■

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 standard [1] 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.

1. C'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.