TD : sémantique opérationnelle (et résolution)

Le langage IML : sémantique opérationnelle structurelle

Exercice 1. Compléter la définition de la relation d'évaluation $\langle b, \sigma \rangle \rightarrow t$, si b est une expression Booléenne de la forme $b_1 \vee b_2$.

Exercice 2. Proposer un commande *cinComm* avec la propriété suivante : pour tout état σ il n'existe pas un état σ' tel que $\langle c, \sigma \rangle \rightarrow \sigma'$. Démontrer cette propriété dans la sémantique.

Induction

Exercice 3. Prouver, par induction structurelle, les propriétés suivantes de la relation d'évaluation des expression arithmétiquesde IML :
– $(a, \sigma) \rightarrow n$ et $(a, \sigma) \rightarrow n'$ implique $n = n'$, pour tout $a \in Aexpr$, tout état σ, et $n, n' \in N$.
– pour tout $a \in Aexpr$ et état σ, il existe $n \in N$ tel que $(a, \sigma) \rightarrow n$.

Exercice 4. Donner des exemples de relations qui sont bien fondées, et de relations qui ne sont pas bien fondé.

Exercice 5. Montrer que le principe d'induction structurelle se réduit à l'induction mathématique. (Suggestion : définir des fonctions de complexité sur les termes).

Exercice 6. Considérer le programme suivant IML :

```
Euclid = while( not (N = M) ) do
  if M <= N then
     N : = N - M
  else
     M : = M - N
```

(qui calcule le PGCD de M et N). Prouver que le programme se termine : donner d'abord un argument intuitif, puis démontrer formellement (dans la sémantique donnée) : $\forall \sigma \neg \exists \sigma'$ t.q. $\langle Euclid, \sigma \rangle \rightarrow \sigma'$.

Un démonstrateur automatique

On rappelle : on veut implémenter en OCaml un démonstrateur automatique dont l'algorithme générique est le suivant :

saturer.code

```
 1 : Procédure saturer(T)
 2 : T : ensemble de clauses
 3 : {
 4 :
 5 :    WO (worked off) (* : ensemble de clauses déjà elaborées *)
 6 :    Us (usable)     (* : enseble de clause utilisables*)
 7 :
 8 :    tant-que( vrai )
 9 :    {
10 :       Us := T
11 :       WO := ensemble vide
12 :
13 :       si Us contient la clause vide
14 :         retourner "théorie contradictoire''
15 :
16 :       si Us est vide,
17 :         retourner "ensemble saturé"
18 :
19 :       choisir C in Us
20 :       ajouter C à WO
21 :       enlever C de Us
22 :       appliquer toutes les règles entre C et WO
23 :         et obtenir un nouvel ensemble de clauses New
24 :       simplifier l'ensemble New (par rapport à New, Us et WO)
25 :       simplifier WO et US (par rapport à new)
26 :       ajouter les clause New à Us
27 :    }
28 : }
```

Exercice 7. Organiser le travail d'implémentation :
- Définir de modules, de sous modules, et des relations de dépendance entre les modules.
- Partager le travail d'implémentation en groupes.

Exercice 8. Rappelons notre implémentation de l'algorithme d'unification :

unify.ml

```
 1 : let rec unify = function
 2 :     [] -> indentite
 3 :   | t::q -> let
 4 :       phi = match t with
 5 :         (v, Var x) | (Var x, v) ->
 6 :           if List.mem x v then failwith "unify"
 7 :           else faire_subst x v
 8 :         | (Operation(f,ff),Operation(g,fg)) ->
 9 :             if( f = g && (List.length ff) = (List.length fg))
10 :             then unify (List.combine ff fg)
11 :             else failwith "unify"
12 :     in
13 :     let
14 :       psi =
15 :       unify (List.map (fun (v1,v2) -> (apply phi v1, apply phi v2)) q )
16 :     in
17 :       compose psi phi
```

Écrire les types et les fonctions (`faire_subst, apply, identite, compose`) qui nous permettent d'utiliser ce code.

Exercice 9. On veut maintenant implémenter les deux règles du calcul de la résolution. Pour cela :
- Proposer des types de donnés pour les formules atomiques et pour les clauses.
- Définir une fonction `unify` dont les arguments sont deux formules atomiques (et qui produit un unificateur de ces formules).
- Définir un fonction `apply` qui applique une substitution à une formule atomique, et une fonction similaire qui applique une substitution à une clause.
- Définir un fonction `fact` qui appliqué à une clause produit la liste de toutes les clauses qu'on peut engendrer à l'aide de la règle de factorisation (droite).
- Définir un fonction `res` qui appliqué à deux clauses produit la liste de toutes les clauses qu'on peut engendrer à l'aide de la règle de résolution.