CHENNI TP4

LINA

Rapport de programmation fonctionnelle

Ce projet traite de l'implémentation de deux fonctions : unif et anti_unif en Ocaml. La fonction unif correspond à l'algorithme d'unification qui prend en entrée deux termes du premier ordre t1 et t2 et renvoie leur unifié. L'algorithme d'anti-unification quant à lui prend en entrée deux termes du premier ordre t3 et t4 et renvoi leur anti-unifié. Pour ce faire je me suis basée sur les algorithmes suivants qui nous ont été fournis en cours et en td:

Algorithme d'unification :

Entrée : un système d'équations E0

Sortie : un système d'équations sous forme résolue équivalent à E0 ou l'exception

ECHEC si E0 n'a pas de solution

E ⟵ E0

Tant que il existe dans E une équation e de la forme :

(1) t = t : suppression de l'équation e

(2) t = v où t est un terme non variable et v est une variable :

remplacement de l'équation e par v = t

(3) F(t1, ..., tn) = G(t'1, ..., t'n) (où F et G sont des métasymboles valant

pour des symboles de fonction) :

Si F et G sont différents

Alors ECHEC

Sinon remplacer l'équation e par les équations

t1= t'1, ..., tn = t'n

(4) v = t où v est une variable qui a au moins une autre occurrence dans E:

Si v apparaît dans t

Alors ECHEC

Sinon garder l'équation e et remplacer toute autre occurrence de v par t

Algorithme d'anti-unification

Fonction Anti_Unif(t,t')

Si t ≡ F(t1, ..., tn) et t' ≡ F(t'1, ..., t'n)

Alors return F(Anti_Unif(t1,t'1), ..., Anti_Unif(tn,t'n))

Sinon Si il existe dans LISTE un couple ((t,t'),j)

Alors return Zj

Sinon générer une nouvelle variable Zi

ajouter le couple ((t,t'),i) à LISTE

return Zi

J'ai choisi de définir un type term qui peut soit être une variable Var dont le nom est donné par une chaîne de caractère, soit être une fonction Func avec un nom de fonction et une liste de termes en arguments. Le type substitution représente une liste de paires associant des variables à des termes. Chaque paire est représentée par une chaîne de caractères (nom de variable) et un terme correspondant.

1) La fonction unif réalise l'unification entre deux termes passés en arguments. Elle vérifie différents cas :

- Si les deux termes sont des variables identiques, elle renvoie une substitution vide et les affiche.
- - Si les deux termes sont des fonctions avec le même nom et la même arité, elle effectue l'unification sur les listes d'arguments. Pour ce faire, la fonction combinaison qui est utilisée pour combiner les termes de même position dans deux listes, prend les deux listes d'arguments et retourne la liste des paires de termes correspondantes. C'est ensuite la fonction uniflist qui prend cette liste en argument et applique l'unification à chaque paire de termes en utilisant la fonction apply. Apply applique une substitution à un terme en utilisant la fonction subst. Elle parcourt la substitution et applique les substitutions successives au terme donné. Pour ce faire, la fonction subst remplace toutes les occurrences d'une variable donnée par un terme dans un autre terme. Elle parcourt récursivement le terme et effectue la substitution lorsque la variable est trouvée. Puis enfin uniflist combine les substitutions obtenues à l'aide de l'opérateur « @ ».
  - Si les deux termes sont des fonctions qui ont un nom différent ou une arité différente, elle renvoie une liste vide et affiche « Non unifiable ».
- Si l'un des termes est une variable et l'autre est une fonction, elle vérifie si la variable apparaît dans la fonction à l'aide de la fonction occurrence qui parcourt

récursivement le terme et renvoie true si la variable est trouvée, sinon false. Si true, elle renvoie une substitution vide et affiche « non unifiable », sinon elle renvoie une substitution avec la paire de variable et de terme correspondante.

2) La fonction anti_unif initialise une liste vide « liste » pour stocker les couples de termes déjà rencontrés. Elle utilise la fonction anti_unif_helper qui prend en arguments deux termes pour effectuer l'anti-unification de manière récursive. Elle vérifie différents cas :

- Si les deux termes sont des fonctions avec le même nom et la même arité, elle effectue l'anti-unification des arguments en appelant récursivement la fonction anti_unif_helper sur chaque paire d'arguments correspondants. Les résultats sont ensuite utilisés pour construire un nouveau terme avec le même nom que les termes d'origine, mais avec les arguments anti-unifiés.

- Si les deux termes sont des fonctions qui ont un nom différent ou une arité différente, la fonction anti_unif_helper recherche dans la liste « liste » si le couple de termes a déjà été rencontré. Si c'est le cas, elle retourne une variable de la forme Var ("Z" ^ string_of_int j), où j est l'indice correspondant au couple de termes trouvé dans la liste. Sinon, elle génère une nouvelle variable de la forme Var ("Z" ^ string_of_int (List.length !liste + 1)), l'ajoute à la liste « liste » avec son index, et retourne cette nouvelle variable.

Concernant les problèmes rencontrés j'avais au début du mal à manipuler correctement les types mais après plusieurs compilations j'ai réussi à corriger mes erreurs. J'ai également rencontré des difficultés pour afficher le résultat de l'unification. En effet avec mon implémentation j'obtiens une liste de paire de substitutions, le terme de gauche correspondant au terme à substituer et le terme de droite correspondant à la substitution à appliquer. Cette difficulté entraîne l'incapacité à mon code de composer avec les deux fonctions directement, par exemple : unifier le résultat de l'anti-unification de t1 et t2 avec le résultat de l'unification de t3 et t4. Toutefois cela demeure possible si on le fait manuellement et si on fait donc tourner le code 2 fois.

Au niveau des jeux d'essais, j'ai essayé mes fonctions avec une fonction et une variable, deux fonctions de même nom et de même arité, deux fonctions de même nom mais pas de même arité, deux fonctions de même arité mais pas de même nom, deux variables égales, deux variables différentes, des fonctions qui ont différentes arités (0,1,2,3). Le code a toujours fonctionné.

Voici quelques exemples pour illustrer cela :

```
# let t1 = Func ("f", [Var "x"; Var "x"; Func ("a", [])]) ;;
val t1 : term = Func ("f", [Var "x"; Var "x"; Func ("a", [])])
# let t2 = Func ("f", [Func ("a", []);  Func ("a", []) ;Var "y"]) ;;
val t2 : term = Func ("f", [Func ("a", []); Func ("a", []); Var "y"])
# let result_unif= unif t1 t2 ;;
val result_unif : substitution =
  [("x", Func ("a", [])); ("y", Func ("a", []))]
# let result_anti_unif= anti_unif t1 t2 ;;
val result_anti_unif : term = Func ("f", [Var "Z1"; Var "Z1"; Var "Z2"])
```

```
# let t1 = Func ("f", [Var "x"; Var "y"; Var "z"]) ;;
val t1 : term = Func ("f", [Var "x"; Var "y"; Var "z"])
# let t2 = Func ("f", [Func ("g", [Var "a"]); Var "y"; Func ("h", [Var "b"; Var
"c"])]) ;;
val t2 : term =
  Func ("f",
   [Func ("g", [Var "a"]); Var "y"; Func ("h", [Var "b"; Var "c"])])
# let result_unif= unif t1 t2 ;;
Unif(y, y) = y
val result_unif : substitution =
  [("x", Func ("g", [Var "a"])); ("z", Func ("h", [Var "b"; Var "c"]))]
# let result_anti_unif= anti_unif t1 t2 ;;
val result_anti_unif : term = Func ("f", [Var "Z1"; Var "Z2"; Var "Z3"])
```

```
# let t1 = Func ("g", [Var "x"; Var "y"; Var "z"]) ;;
val t1 : term = Func ("g", [Var "x"; Var "y"; Var "z"])
# let t2 = Func ("f", [Func ("g", [Var "a"]); Var "y"; Var "c"]) ;;
val t2 : term = Func ("f", [Func ("g", [Var "a"]); Var "y"; Var "c"])
# let result_unif= unif t1 t2 ;;
Non unifiable
val result_unif : substitution = []
# let result_anti_unif= anti_unif t1 t2 ;;
val result_anti_unif : term = Var "Z1"
```