Prog. Fonctionnelle : rapport du devoir

Membres du groupe :

Dieunel MARCELIN 12217041
Amazigh ALLOUN 12007813

Introduction :

Le but de ce rapport est de présenter les fonctions *unif* et *anti_unif* et l'ensemble des structures de données utilisées pour les implémenter en OCaml. Les fonctions unif et anti_unif sont des algorithmes courants utilisés en programmation logique pour unifier ou anti-unifier des termes du premier ordre. Pour représenter les termes du premier ordre, nous avons utilisé une structure *term* qui est soit une variable *Var* ou une constante *const* de type string chacune, ou alors une fonction *Func* dont le nom est un string et qui contient une liste de ses paramètres. Nous avons choisi cela car c'était la représentation la plus efficiente et simple à manipuler. Nous avons également choisi de représenter les substitutions avec le type *subst* sous forme d'une liste de paires 'variable termes'.

La fonction unif et son implémentation :

La fonction *unif* du programme prend en paramètre deux termes du premier ordre et renvoie leur unifié sous forme d'une liste de couple où le premier élément est de type *Var* et le second est un *term*, elle lève une exception en cas d'échec. Cette fonction fait appel à d'autres fonctions intermédiaires : *occurs_check* qui prend une variable et un terme et renvoie true si le terme contient la variable et false sinon ; *substitute_term* qui prend un term et une substitution et qui permet de substituer des variables dans ce term à l'aide de la subst donnée en paramètre ; et enfin la fonction *unify_args* qui prend deux arguments de deux term (la liste des arguments), tente de les unifier en faisant appel à *substitute_term* et renvoie le résultat, sinon l'exception *FAILURE* en cas d'erreur.

La fonction anti_unif et son implémentation :

La fonction *anti_unif* a la même entrée que *unif*, mais renvoie quant à elle l'anti-unifié des deux termes passés en paramètre, ou bien évidemment *FAILURE* en cas d 'erreur. Elle calcule, dans le cas où les termes sont deux fonctions de même nom et arité, récursivement l'anti_unifié des arguments de chaque fonction un-à-un et renvoie le terme avec en argument la liste des anti-unifiés. Nous traiterons des autres cas en détail plus loin.

Difficultés rencontrées :

La difficulté majeure que nous avons rencontrée était le manque de maîtrise de OCaml, étant récemment exposés à ce langage ce qui a fait que nous n'étions pas à l'aise avec la syntaxe. De ce fait nous avons dû effectuer plusieurs recherches pour trouver les bonnes terminologies et corriger les bugs qui résulter de cette lacune, principalement des erreurs de typage.

Commentaire du code :

Ayant déjà traiter les types implémentés ci-dessus, nous allons parler directement des fonctions et choix d'implémentation.

* unif :

Cette fonction prend deux termes t1 et t2 en entrée.

• Si t1 est une variable (i.e. Var x) :

Elle vérifie d'abord si t1 est égal à t2 en utilisant l'expression si t1 = t2. Si tel est le cas, elle renvoie une liste vide [] car aucun remplacement n'est nécessaire. Sinon, elle utilise la fonction *occurs_check* pour vérifier si t2 contient l'occurrence de la variable x. Si c'est le cas, cela signifie qu'il existe une dépendance entre les variables et qu'elle ne peut pas être unifiée. Dans ce cas, *unif* lève l'exception *FAILURE*. Sinon, celle-ci renvoie une substitution contenant le couple (x, t2), ce qui signifie que la variable x est remplacée par le terme t2.

• Si t2 est une variable (i.e. Var x):

Très similaire au cas précédent, la seule différence est que l'ordre est inversé donc on inverse également les rôles de t1 et t2.

• Si t1 et t2 sont des constantes (Const c1 et Const c2) :

Il vérifie si les constantes c1 et c2 sont les mêmes (c1 = c2). Si tel est le cas, la liste vide [] est renvoyé. Sinon , l'exception FAILURE sera levée comme ils ne peuvent pas être fusionnés.

• Si les deux termes sont des fonctions avec des noms de fonction identiques, elle fait appel à *unify_args* qui est appliqué aux listes d'arguments de la fonction.

• Si aucune des circonstances précédentes n'est satisfaite, une exception *FAILURE* sera déclenchée pour indiquer que l'unification ne peut avoir lieu.

* unify_args:

On appelle cette fonction dans *unif* et lui passe en paramètres les arguments des deux fonctions (quand c'est bien le cas) :

• Si les deux listes d'arguments ne contiennent aucun élément, une substitution vide est renvoyée.

• Si seulement une des deux listes d'arguments est vide, une exception FAILURE est déclenchée.

• Si les deux listes d'arguments ne sont pas vides, *unif* est appliquée au premier argument de chaque liste, puis récolte *unify_args* au reste des listes d'arguments en utilisant la substitution. Ensuite, il combine les résultats à la place.

* substitute_term:

Elle effectue la substitution des variables dans un terme donné en utilisant une substitution fournie.

• Si le terme est une variable, elle recherche un remplacement pour la paire associée à cette variable et renvoie le terme correspondant. Si la variable n'est pas dans le remplacement, elle renverra le terme inchangé.

• Si le terme est une fonction, il applique la substitution aux arguments de chaque fonction en utilisant la récursivité et renvoie une nouvelle fonction avec les arguments remplacés.

• Dans tous les autres cas, elle renvoie le terme inchangé.

*anti_unif :

Elle trouve un terme généralisé qui englobe deux termes donnés en paramètre. Elle effectue une généralisation récursive des paires d'arguments correspondants et génère des variables uniques pour les parties non unifiables des termes.

• Si les deux termes t1 et t2 sont des fonctions avec deux noms distincts ou que les fonctions ne prennent pas le même nombre d'arguments, l'exception *FAILURE* est levée.

• Si les deux termes t1 et t2 sont des fonctions avec le même nom de fonction (Func (f1, args1) et Func (f2, args2)), la fonction anti_unif est appelée récursivement sur les listes d'arguments correspondantes args1 et args2. Cela permet de trouver les termes généralisés pour chaque paire d'arguments correspondants.

• Si une paire de termes ne peut pas être anti-unifiée, c'est-à-dire qu'ils n'ont pas le même nom de fonction, ou que les paires d'arguments ne peuvent pas être anti-unifiées, la fonction génère une nouvelle variable unique (Var ("Z" ^ string_of_int i)) où i est un compteur qui maintient le nombre de variables générées. Cette variable représente la généralisation de la paire de termes non unifiables

Quelques exemples à tester : (sous environnement *OCaml* ou *utop*)

Sous linux, ouvrez un terminal dans le répertoire du fichier « devoir_pf_final.ml », tapez ocaml ou utop au clavier et enfin la commande : #use "devoir_pf_final.ml";;

unif :

1) unif (Var "x") (Const "d");; (* affiche : (string * term) list = [("x", Const "d")]*)
2) unif (Var "x") (Var "x");; (* affiche : (string * term) list = []*)
3) unif (Func ("f", [Var "x"])) (Func ("g", [Var "z"]));; (* affiche exception : FAILURE*)
4) unif (Func ("f", [Var "a"; Var "y"])) (Func ("f", [Const "w"; Var "z"]));;
   (* affiche (string * term) list = [("y", Var "z"); ("a", Const "w")] *)
5) unif (Func ("f", [Var "x"; Var "y"])) (Func ("f", [Var "z"; Func("g", [Var"k"])]));;
   (* affiche (string * term) list = [("y", Func ("g", [Var "k"])); ("x", Var "z")] *)
6) unif (Func ("f", [Var "a"; Var "y"; Var"v"])) (Func ("f", [Const "w"; Var "z"; Const"b"]));;
   (* affiche (string * term) list = [("v", Const "b"); ("y", Var "z"); ("a", Const "w")]*)

anti_unif : (recompilez le programme)

1) anti_unif (Var "x") (Const "d");; (* affiche term = Var "Z0"*)
2) anti_unif (Var "x") (Var "x");; (* affiche term = Var "Z1"*)

3) anti_unif (Func ("f", [Var "x"])) (Func ("g", [Var"a"; Var "z"]));; (*affiche Exception:FAILURE*)

4) anti_unif (Func ("f", [Var "a"; Var "y"; Var"v"])) (Func ("f", [Const "w"; Var "z"; Const"b"]));;
(* affiche term = Func ("f", [Var "Z3"; Var "Z4"; Var "Z5"])*)

5) anti_unif (Func ("f", [Var "a"; Var "y"; Var"v"])) (Func ("f", [Var "w"; Var "z"]));;
(* affiche exception : FAILURE *)