Rapport de Projet LMC

# But du projet

Le but de ce projet est d’implanter en PROLOG une variante de l’algorithme d’unification de Martelli-Montanari vu en cours.

# Description du code

Predicats d’affichage fournis

set\_echo :- assert(echo\_on).

Ce prédicat active l’affichage par le prédicat echo

clr\_echo :- retractall(echo\_on).

Ce prédicat inhibe l'affichage par le prédicat echo.

echo(T)

Si le flag echo\_on est positionné, echo(T) affiche le terme T sinon, echo(T) réussit simplement en ne faisant rien.

echo(T) :- echo\_on,

!,

write(T).

echo(\_).

print(Term) :- current\_prolog\_flag(print\_write\_options, Options),

!,

write\_term(Term, Options).

Permet d’afficher un terme.

Les prédicats

occur\_check(V,T) :- V==T -> !;

compound(T),

functor(T,N,A),

ocheck(V,T,A), !.

Teste si V est égal à T, s’il l’est, s’arrête là.

Teste si T est un terme composé. Si c’est le cas, appelle ocheck(V,T,A).

ocheck(V,T,A) :- A==1 ->arg(1,T,X),

occur\_check(V,X) ;

arg(A,T,X) ;

occur\_check(V,X) ;

A2 is (A-1),

ocheck(V,T,A2), !.

\*On vérifie que A est égal à 1, (donc un seul argument), si c’est le cas, on appelle une seule fois occur\_check sur V et X.

Applique de manière récursive occur\_check sur V et sur tous les arguments de T.

regle(E,R)

Détermine la règle de transformation R qui peut s’appliquer à l’équation E

regle((\_ ?= T),rename) :- var(T),!.

Vérifie si T est une variable afin de pouvoir appliquer la règle rename si c’est le cas

regle((X ?= T), clash) :- compound(X),

compound(T),

functor(X,\_,N1),

functor(T,\_,N2),

N1 \== N2,!;

compound(X),

compound(T),

functor(X,A1,\_),

functor(T,A2,\_),

A1 \== A2,!.

Vérifie que X et T ne sont pas des termes composés et qu’ils ont le même nombre d’arguments (N1 \== N2,!;)Si c’est le cas, applique la règle clash

Ou vérifie que X et T ne sont pas des termes composés et n’ont pas le même nom(A1 \== A2,!.) Si c’est le cas, applique la règle clash.

regle((\_ ?= T),simplify) :- atomic(T),!.

Vérifie que T est une constante. Si c’est le cas, applique la règle simplify.

regle((X ?= T), check) :- X \== T,

occur\_check(X,T),!.

Vérifie que X est différent de T et que X apparait dans T. Si c’est le cas, applique la règle check.

regle((\_ ?= T),rename) :- var(T),

!.

Vérifie que T est une variable. Si c’est le cas, applique la règle rename.

regle((X ?= T), expand) :- compound(T),

not(occur\_check(X,T)),

!.

Vérifie que T est un terme et aussi que X n’apparait pas dans T. Si toutes ces conditions sont réunies, la règle expand est appliquée.

regle((X ?= T), decompose) :- compound(X),

compound(T),

functor(X,A,N),

functor(T,A,N),

!.

Vérifie que X et T sont des termes composés et qu’ils ont le même nom et le même nombre d’arguments. Si ces conditions sont réunies, la règle decompose est appliquée.

regle((T ?= \_), orient) :- nonvar(T),

!.

Vérifie que T n’est pas une variable et que X est une variable. Si c’est le cas, applique la règle orient.

reduit(rename,(X ?= T), P;S, A;[X = T|B]) :- echo(system : [X ?= T|P]), nl,

echo(rename : X ?= T),

nl,

substitution(X,T,P,A), substitution(X,T,S,B).

reduit(simplify,(X ?= T), P;S, A;[X = T|B]) :- echo(system : [X ?= T|P]), nl,

echo(simplify : X ?= T),

nl,

substitution(X,T,P,A),

substitution(X,T,S,B).

reduit(check,\_,\_,\_) :- echo(system : [X ?= T|P]), nl,

echo(check : X ?= T),

nl,

fail.

reduit(expand,(X ?= T), P;S, A;[X = T|B]) :- echo(system : [X ?= T|P]), nl,

echo(expand : X ?= T),

nl,

substitution(X,T,P,A), substitution(X,T,S,B).

reduit(orient,(X ?= T), P;S, [T ?= X|P];S) :- echo(system : [X ?= T|P]), nl,

echo(orient : X ?= T),

nl.

reduit(decompose,(X ?= T), P;S, Q;S) :- echo(system : [X ?= T|P]), nl,

echo(decompose : X ?= T),

nl,

decomposer(X,T,L), concat(L,P,Q).

reduit(clash,\_,\_,\_) :- echo(system : [X ?= T|P]), nl,

echo(clash : X ?= T),

nl,

fail.

reduit(R,E,P,Q) : Transforme le système d'équations P en le système d'équations Q par application de la règle de transformation R à l'équation E.

substitution(\_,\_,[],[]).

substitution(X,T,[A ?= B|P],[A2 ?= B2|P2]) :- substitution\_terme(X,T,A,A2),

substitution\_terme(X,T,B,B2), substitution(X,T,P,P2).

substitution(X,T, [ A = B |P], [A2 = B2 | P2]) :- substitution\_terme(X,T,A,A2), substitution\_terme(X,T,B,B2),

substitution(X,T,P,P2).

Remplace le terme A par T si A est égal à X. Si ce n'est pas le cas, et que A n'est pas composé, rien n'est remplacé. Si A n'est pas égal à X mais que A est tout de même composé, on remplace X par T.

substitution\_terme(X,T,A,T) :- A==X.

substitution\_terme(X,\_,A,A) :- not(compound(A)),

A\==X.

substitution\_terme(X,T,A,B) :- A\==X,

compound(A),

substitutionSousTerme(X,T,A,B).

Transforme une liste d'équations (3ème argument) en remplaçant X par T, pour retourner le résultat dans une liste modifiée (4ème argument) avec les valeurs remplacées

substitutionSousTerme(X,T,A,B) :- functor(A,N,\_),

creerListe(X,T,A,0,L),

B =..[N|L].

Crée le terme B à partir de A où l'on remplace les X par T.

creerListe(\_,\_,F,N,[]) :- functor(F,\_,A),

(N>=A).

On remplace directement X par T.

creerListe(X,T,F,N,[E|L]) :- N2 is N+1,

arg(N2,F,E),

(E\==X),

not(compound(E)),

creerListe(X,T,F,N2,L).

Lorsque E n'est pas composé.

creerListe(X,T,F,N,[G|L]) :- N2 is N+1,

arg(N2,F,E),

(E\==X),

compound(E),

substitutionSousTerme(X,T,E,G),

creerListe(X,T,F,N2,L).

E est composé, on rappelle substitutionSousTerme sur E pour y substituer les X aux T.

creerListe(X,T,F,N,[T|L]) :- N2 is N+1,

arg(N2,F,E),

(E==X),

creerListe(X,T,F,N2,L).

On avance dans la liste.

decomposer(T,X,L) :- decomposition(T,X,0,L).

decomposition(T,\_,N,[]) :- functor(T,\_,N).

decomposition(T,X,N,[A ?= B|L]) :- N2 is N+1,

arg(N2,T,A),

arg(N2,X,B),

decomposition(T,X,N2,L).

A partir d'une équation de la forme f(X1,...,Xn) ?= f(Y1,...,Yn), on décompose élément par élément l'équation pour renvoyer une liste d'égalités de la forme [X1 ?= Y1, ... , Xn ?= Yn].

concat([],X,X).

concat([X|P],Y,[X|Q]) :- concat(P,Y,Q).

Concaténation de deux listes.

ident([]).

ident([ X = X | T]) :- ident(T).

Force l'identification => c'est ce qui permet à la fin d'afficher les valeurs des variables et non le nom que leur a donné le système

poidRegle(clash,5).

poidRegle(check,5).

poidRegle(rename,4).

poidRegle(simplify,4).

poidRegle(orient,3).

poidRegle(decompose,2).

poidRegle(expand,1).

Poids de chaque règle selon l'énoncé

choix\_pondere([H|T],Q,F,R) :- regle(H,RE),

poidRegle(RE,W),

selectEqu1(T,Q,F,H,W,R,RE).

A partir d'une liste d'équations, on renvoie l'équation dotée de la règle de plus haut poids

et la liste des équations restantes (non triées) ainsi que la règle associée.

liste paramètres:

1 : liste d'équations de départ

2 : liste d'équations finales

3 : équation résultat

4 : var de travail pour l'équation

5 : var de travail pour le poids

6 : résultat règle

7 : var de travail regle

selectEqu1([],[],E,E,\_,R,R) :- !.

Si on a déjà trouvé une règle de poids maximal, on arrête le parcours de la liste.

selectEqu1(Q,Q,E,E,W,R,R) :- W >= 5,

!.

selectEqu1([H|T],[H|Q],F,E,M,S,R) :- regle(H,RE),

poidRegle(RE,W),

(W =< M),

selectEqu1(T,Q,F,E,M,S,R).

selectEqu1([H|T],[E|Q],F,E,M,S,\_) :- regle(H,RE),

poidRegle(RE,W),

(W > M),

selectEqu1(T,Q,F,H,W,S,RE).

choix\_premier([E|P],P,E,R) :- regle(E,R).

Fonction choix\_premier : renvoi simplement la règle qui s'applique à la première équation du système.

unifie(P,S) :- set\_echo, sousUnifie(P;[],S).

sousUnifie([];S,\_) :- ident(S).

sousUnifie(P;S,choix\_premier) :- choix\_premier(P,Z,E,R), reduit(R,E,Z;S,Q), sousUnifie(Q,choix\_premier).

sousUnifie(P;S,choix\_pondere) :- choix\_pondere(P,Z,E,R), reduit(R,E,Z;S,Q), sousUnifie(Q,choix\_pondere).