Mahr Benjamin
Ginestra Alex
Logique et Modèle de Calculs

Rapport: projet LMC

Dans ce rapport, nous allons étayer la manière dont nous avons conçu notre programme prologue mais également répondre aux questions du sujet donné.

Liste des prédicats fournis:

• set_echo: (Prédicat active l'affichage par le prédicat echo)

set_echo :- assert(echo_on)

• clr_echo: (Prédicat inhibe l'affichage par le prédicat echo)

clr_echo :- retractall(echo_on)

Si le flag **echo_on** est positionnée alors on affiche le terme « **T** ».

echo(T):- echo_on, !, write(T).

echo(_).

Questions n°1:

• Prédicat **occur_check** : Teste si V apparaît dans le terme T.

Si le terme V est égal au terme T le prédicat est vrai sinon il retourne faux:

occur_check(V,T) :- V == T, !.

Si le terme T est une fonction, on récupère son arité par la fonction **functor(T,_,A)** et on regarde si V est présent parmi ses arguments, avec le prédicat **occur_check_base(V,T,A)**:

occur_check(V,T) :- compound(T), functor(T,_,A), occur_check_base(V,T,A).

• Prédicat **occur_check_base**: Teste si V est un des arguments de la fonction T :

Si N est égale à 1 alors on récupère la valeur du premier argument de T dans une variable X.

Et on appelle **occur_check** avec V et X en paramètre, qui sera vrai si V == X.

occur_check_base(V,T,1) :- arg(1,T,X),!,occur_check(V,X).

Si N est différent de 1 alors on récupère la valeur du N-ième (dernier) argument dans une variable X. Ensuite on appelle occur_check avec V et X en paramètre, qui sera vrai si V = X

ou si V est présent dans les arguments de X dans le cas ou X est une fonction. Ensuite on appelle récursivement le prédicat en décrémentant N pour tester les autres argument de T.

```
occur_check_base(V,T,N) :- arg(N,T,X),occur_check(V,X);N1 is
(N-1),occur_check_base(V,T,N1).
```

Liste des règles:

Ces prédicats sont vrais si nous pouvons appliquer la règle (deuxieme argument) sur l'équation (premier argument).

- Règle Rename:

  Ce prédicat est vrai si nous pouvons appliquer la règle rename sur l'équation E (si T est une variable).

```
regle((_ ?= T),rename) :- var(T),!.
```

- Règle Simplify:

  Ce prédicat est vrai si nous pouvons appliquer la règle simplify sur l'équation E (si T est une constante).

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

- Règle Orient:

  Ce prédicat est vrai si nous pouvons appliquer la règle orient sur l'équation E (si T n'est pas une variable).

```
regle((T ?= _),orient) :- nonvar(T),!.
```

- Règle Check:

  Ce prédicat est vrai si nous pouvons appliquer la règle check sur l'équation E (si X est différent de T et X apparaît dans T).

```
regle((X ?= T),check) :- X \== T, var(X), occur_check(X,T),!.
```

- Règle Expand:

  Ce prédicat est vrai si nous pouvons appliquer la règle expand sur l'équation E (si X est une variable,T est une fonction et que X n'apparaît pas dans les arguments de T).

```
regle((X ?= T),expand) :- var(X), compound(T), not(occur_check(X,T)), !.
```

- Règle Decompose:

  Ce prédicat est vrai si nous pouvons appliquer la règle décompose sur l'équation E (si les deux termes de l'équation sont des fonctions et que leurs noms sont égaux ainsi que leur arité, en effet on récupère leurs noms et leurs arités par le prédicat **functor()** et on teste si ils sont égaux.)

```
regle((X ?= T),decompose) :- compound(X), compound(T), functor(X,A1,N1),
functor(T,A2,N2),A1==A2,N1==N2,!.
```

• Règle Clash:

Ce prédicat est vrai si nous pouvons appliquer la règle clash sur l'équation E (si X et T sont des fonctions et que leurs arités sont différentes, on récupère leurs arités par le prédicat **functor()**).

```
regle((X ?= T),clash) :- compound(X), compound(T), functor(X,_,N1),
                         functor(T,_,N2), N1 \== N2, !.
```

Ce prédicat est vrai si nous pouvons appliquer la règle clash sur l'équation E (si X et T sont des fonctions et que leur noms sont différent, en effet on récupère leurs noms par le prédicat **functor()**).

```
regle((X ?= T),clash) :- compound(X), compound(T), functor(X,A1,_),
                         functor(T,A2,_), A1 \== A2, !.
```

Prédicats utiles pour création des prédicats réduits:

• append: (prédicat qui concatène les deux premiers arguments dans le troisième)

Ce prédicat s'arrête (ne fais rien) quand le deuxième argument est un tableau vide.

```
append(X,[],X).
```

ce prédicat concatène les deux premiers arguments dans le troisième et appelle récursivement le prédicat **append()**.

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

• substitution : (prédicat qui substitue toutes les occurrences du premier argument par le deuxième argument de l'équation du troisième argument et place ce résultat dans le quatrième argument)

Si le troisième argument et le quatrième argument sont des tableaux vides alors on ne fait rien.

```
substitution(_,_,[],[]) :- !.
```

On réalise les substitution des différentes parties de l'équations du système en appelant le prédicat substitution terme. Puis on appelle récursivement substitution(X,T,P,P2), reste de l'équation à modifier en remplaçant X par T.

```
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_terme: (remplace le premier argument/terme par le deuxième argument/terme):

Si A n'est pas un terme composé et que A est égale a X, prédicat vrai.

```
substitution_terme(X,T,A,T):- A == X, not(compound(A)), !.
```

Si a n'est pas un terme composé et que A est différent de X, prédicat vrai.

substitution_terme(X, _,A,A):- A \== X, not(compound(A)), !.

Si A est un terme composé alors on appelle récupère l'arité de la fonction par **functor()** et on appelle **substitution_funct()** avec l'arité de la fonction, ce prédicat va alors transformer les arguments de A égaux à X par T. Le prédicat place le résultat obtenu dans le paramètre Q.

substitution_terme(X,T,A,Q):- functor(A,_,N), compound(A), substitution_funct(X,T,A,N,Q), !.

- substitution_funct: (prédicat qui substitue les occurrences de X par T qui se situe dans l'équation A et place ce résultat dans Q)

  Si l'arité de la fonction est égale à 1, on récupère le nom de la fonction A dans F , l'arité de A dans N et on récupère ensuite la valeur du premier argument de la fonction A dans la variable B, on récupère le nom de la fonction Q qui doit être égal à F et l'arité de Q qui doit être égal à N et on récupère la valeur du premier argument de la fonction Q dans la variable V on appelle ensuite le prédicat substitution_terme(X,T,B,V).

substitution_funct(X,T,A,1,Q):- functor(A,F,N), arg(1,A,B), substitution_terme(X,T,B,V), functor(Q,F,N), arg(1,Q,V), !.

  Si l'arité est supérieur à 1, on récupère le nom de la fonction A dans F et son arité dans M.Ensuite on récupère la valeur du dernier argument dans la variable B. On récupère le nom de la fonction Q qui doit être égal à F et son arité qui doit être égal à M. Ensuite on récupère la valeur du dernier argument de Q dans la variable V. On appelle ensuite le prédicat substituion_terme(X,T,B,V) puis enfin on appelle récursivement ce prédicat tant que N est supérieur à 1.

substitution_funct(X,T,A,N,Q):- functor(A,F,M), arg(N,A,B), substitution_terme(X,T,B,V), functor(Q,F,M), arg(N,Q,V), N2 is (N-1), substitution_funct(X,T,A,N2,Q),!.

- Substitution_autre: (prédicat qui va substituer toutes les occurrences du premier argument par le deuxième argument de l'équation du troisième argument et place se résultat dans le quatrième argument)

  Si le troisième argument et le quatrième argument sont des tableaux vides alors on ne fait rien.

substitution_autre(_,_,[],[]):- !.

  On réalise les substitution des différentes parties de l'équations du système en appelant le prédicat substitution terme. Puis on appelle récursivement substitution(X,T,P,P2), reste de l'équation à modifier en remplaçant X par T.

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

substitution_terme(X,T,A,A2),substitution_terme(X,T,B,B2), substitution_autre(X,T,P,P2).

- decomposer: (décompose élément par élément l'équation passée en paramètre (premier argument) par rapport à l'arité de la fonction de départ (deuxième argument). On place dans L2 la liste les égalités de fin obtenu)

  On récupère la valeur du premier argument de X dans A et de T dans B et on place l'égalité A ?= B et L1 dans L2 (quatrième argument).

  decomposer((X ?= T),1,L1,[A ?= B|L1]) :- arg(1,X,A), arg(1,T,B), !.

  On récupère la valeur du dernier argument de X dans A et de T dans B et puis on appelle récursivement decomposer(X?=T, N2, [A?=B |L1], L2) en décrémentant N dans N.

decomposer((X ?= T),N,L1,L2) :- N2 is (N-1), arg(N,X,A), arg(N,T,B), decomposer(X ?= T,N2,[A ?= B|L1],L2).

Prédicats réduits :

- réduit: (prédicat qui transforme le système d'équation (troisième argument) en système d'équation stocké dans quatrième argument par application de la règle (premier argument) sur le système d'équation du deuxième argument.)

reduit(decompose,(X ?= Y),P1;Q,P2;Q):- echo(system :[X = Y|P1]), echo('\n'), echo(decompose : (X = Y)), echo('\n'), functor(X,_,A), decomposer(X ?= Y,A,[],L), append(P1,L,P2),!.

reduit(rename,(X ?= Y),P1;Q1,P2;[X=Y|Q2]):- echo(system :[X = Y|P1]), echo('\n'),echo(rename :(X = Y)),echo('\n'), substitution(X,Y,P1,P2),substitution_autre(X,Y,Q1,Q2),!.

reduit(simplify,(X ?= Y),P1;Q1,P2;[X=Y|Q2]):- echo(system :[X = Y|P1]), echo('\n'),echo(simplify :(X = Y)),echo('\n'), substitution(X,Y,P1,P2),substitution_autre(X,Y,Q1,Q2),!.

reduit(expand,(X ?= Y),P1;Q1,P2;[X=Y|Q2]):- echo(system :[X = Y|P1]), echo('\n'), echo(expand :(X = Y)), echo('\n'), substitution(X,Y,P1,P2), substitution_autre(X,Y,Q1,Q2),!.

reduit(orient,(X ?= Y),P;Q,[X ?= Y|P];Q):- echo(system :[X = Y|P]), echo('\n'),echo(orient :(X = Y)),echo('\n'),!.

reduit(check,(X ?= Y),P;Q,P;Q):- echo(system :[X = Y|P]),echo('\n'),echo(check :(X = Y)),echo('\n'),write('\n No'),fail,!.

reduit(clash,(X ?= Y),P;Q,P;Q):- echo(system :[X = Y|P]),echo('\n'),echo(clash :(X = Y)),echo('\n'),write('\n No'),fail,!.

Questions n°2:

• poids : (prédicat qui défini l'ordre de priorité des règles)

```
poids(clash,5).
poids(check,5).
poids(rename,4).
poids(simplify,4).
poids(orient,3).
poids(decompose,2).
poids(expand,1).
```

• retirerElem: (prédicat qui retire le premier argument de l'équation du deuxième argument et place le reste dans le troisième argument.)

Si le deuxième et troisième sont vide, on ne fais rien.

```
retirerElem(_,[],[]):- !.
```

Si X est égal à T alors V prend la valeur R, prédicat est vrai sinon X est différent de T alors on appelle récursivement retirerElem(X,R,V).

```
retirerElem(X,[T|R],V):- ((X == T)->(V = R),!; retirerElem(X,R,V)).
```

• ordrePoids: (prédicat cherche le prédicat règle a appliquer par rapport à l'équation et au poids de chacune des règles)

On applique la règle(X,R).

```
ordrePoids([X],R,X):- regle(X,R), !.
```

On applique la règle R1 sur l'équation X, on récupère le poids de R1. On applique la règle R2 sur l'équation T, on récupère le poids de R2. Si P1 >= P2 alors on appelle récursivement ordrePoids([T|P],R,E) sinon P1=< P2 alors on appelle récursivement ordrePoids([X|P],R,E).
.

```
ordrePoids([X,T|P],R,E):- regle(X,R1),poids(R1,P1),regle(T,R2),poids(R2,P2),((P1 >= P2)->ordrePoids([X|P],R,E);ordrePoids([T|P],R,E)).
```

Ce prédicat appelle le prédicat echo pour sauter une ligne ici.

```
println([]):- !.
```

Ce prédicat affiche l'équation X = T avec les valeurs de X et T et appelle la suite des résultats à écrire par println(P).

```
println([X=T|P]):- echo(X = T),echo('\n'), println(P).
```

• choix_premier: (réalise la stratégie choix_premier)

ce prédicat appelle le prédicat println(Q) et écrit Yes pour signaler que le système est unifiable.

```
choix_premier([],Q,_,_):- echo('\n'), println(Q), echo('\n'), write('Yes'),!.
```

On applique le prédicat règle(E,R) et le prédicat reduit(R,E,P;Q,P2;Q2) et on appelle le récursive ment le prédicat choix_premier(P2,Q2_,_) pour réaliser la suite des règles et réduits du système d'équation.

choix_premier([E|P],Q,E,R):- regle(E,R), reduit(R,E,P;Q,P2;Q2), choix_premier(P2,Q2,_,_).

- choix_pondere: (réalise la stratégie choix_pondere)

  ce prédicat appelle le prédicat println(Q) et écrit Yes pour signaler que le système est unifiable.

  choix_pondere([],Q,_,_):- echo('\n'),println(Q),echo('\n'),write('Yes'),!.

  On applique le prédicat ordrePoids(P1,R,E) pour savoir quel règle est à appliquer en premier, puis on retirerElement(E,P1,P2) et on appelle le prédicat réduit(R,E,P2;Q,P3;Q3) puis on appelle récursivement choix_pondere(P3,Q3,_,_) pour réaliser les autres règles et réduits à appliquer sur le système d'équations.

  choix_pondere(P1,Q,E,R):- ordrePoids(P1,R,E), retirerElement(E,P1,P2),
  reduit(R,E,P2;Q,P3;Q3), choix_pondere(P3,Q3,_,_).

- unifie: (prédicat qui appelle la stratégie passé en second paramètre pour l'appliquer sur le système d'équation passé en premier paramètre. Si ce prédicat est appellé avec seulement en paramètre le système d'équation, la stratégie par défaut qui sera appliqué est le choix_premier.

  unifie(P, premier):- choix_premier(P,_,_,_).
  unifie(P, pondere):- choix_pondere(P,_,_,_).
  unifie(P):- choix_premier(P,_,_,_).

Questions n°3:

- trace_unif: (Prédicat qui appelle le prédicat set_echo pour faire apparaitre les traces d'execution ensuite appelle unifie(P,Strategie) et enfin appelle clr_echo pour inhiber l'affichage)

  trace_unif(P,Strategie):- set_echo,unifie(P,Strategie),clr_echo,!.

- clr_echo: (Prédicat qui appelle le prédicat clr_echo pour inhiber l'affichage des traces d'exécution. Puis appelle unifie(P,Strategie) et enfin appelle une nouvelle fois clr_echo)

  unif(P,Strategie):- clr_echo,unifie(P,Strategie),clr_echo, !.

Amelioration Interface:

- trace_unif: (Prédicat qui appelle trace_unif(P,Strategie) si le troisième paramètre est oui cela veut dire que l'utilisateur veut voir les traces d'exécution)

  instruction(P,Strategie,oui) :- trace_unif(P,Strategie), !.

- trace_unif: (Prédicat appelle trace_unif(P,Strategie) si le troisième paramètre est non cela veut dire que l'utilisateur ne veut pas voir les traces d'execution)

instruction(P,Strategie,non) :- unif(P,Strategie), !.

- option:

  Prédicat qui appelle démarrer. pour utiliser à nouveau l'application du programme de l'algorithme de Martelli-Montanari si l'utilisateur souhaite l'utiliser.

  option(oui):- demarrer.

  Prédicat renvoie fail si si l'utilisateur ne souhaite pas utiliser à nouveau le programme.

  option(non):- fail, !.

- fin: (Prédicat affiche et demande à l'utilisateur si il souhaite à nouveau utiliser l'algorithme de Martelli-Montanari)

  fin:- write('\n \t Souhaitez-vous continuer a utiliser le programme? Entrer oui ou non \n'), write(' \t Choix:'), read(Choix), option(Choix), !.

- demarrer: (Prédicat affiche et demande à l'utilisateur quel système d'équations est à soumettre au programme de Martelli-Montanari, quelle stratégie est à choisir et enfin demande à l'utilisateur s'il souhaite afficher les traces d'exécution)

  demarrer:- write('\t Ce programme repose sur l\'Algorithme d\'unification de Martelli-Montanari \n \n Nb: N\'oubliez pas d\'ajouter un point a chaque entre \n'), write('\n Ecrire le systeme que vous souhaitez unifier: \n'), write(' \t Systeme equation:'), read(Sys), write('\n Differents choix de strategie: laquelle desirez-vous utiliser ? Entrez premier ou pondere \n'),write('\t Strategie:'), read(Strat), write('\n Souhaitez-vous faire apparaitre dans le terminal la trace ? Entrer oui ou non \n'), write('\t Choix:'), read(Choix) ,write('\n'), instruction(Sys, Strat, Choix),fin, !.

A la fin de notre programme les valeurs des variables X, Y, Z,… s'affiche en prenant la valeur que le système leur a donné.