Université de Lorraine

Faculté de Science et Technologie

Master 1 Informatique

Projet de LMC

Algorithme d'unification

Martelli Montanari

Auteurs :
Baptiste Lesquoy
Nicolas Weissenbach

15 décembre 2015

Table des matières

Question 1

Les premiers pas vers l'unification

Dans un premier temps, nous avons décidé d'écrire tous les prédicats de règle (qui permettent de savoir quelle règle utiliser) et les prédicats de réduction. Commencer par ces prédicats aura permis d'avoir une base de travaille et d'emprunter un processus de développement itératif.

Exemple d'un prédicat *regle*

```
regle(E, simplify):-
  split(E, L, R),
  not(var(R)),
  var(L),
  not(compound(R))
.
```

Listing 1 – Choix de la règle $simplify$

Ici nous avons le prédicat regle pour l'unification avec $simplify$, le but de ce prédicat est de renvoyer vrai si simplify est applicable à une equation E. Le prédicat est faux si la règle n'est pas applicable.
Toutes les règles on été instanciées pour chacune des unifications possible.

Exemple d'un prédicat *reduit*

```
reduit(expand, E, P, Q) :-
  split(E, X, T),
  X = T,
  delete_elem(E, P, Q)
.
```

Listing 2 – Application de la réduction *expand*

Le prédicat *reduit* permet d'appliquer la réduction approprié à l'équation E. Ici le prédicat *reduit* applique la transformation *expand* à l'équation E et crée Q le nouvel ensemble de travail.

Unification

A la suite de l'implémentation de ces prédicats, nous avons mis en place le système de prédicat qui permet résoudre l'unificateur le plus général. Pour cela, nous avons implémenté le prédicat $unifie$. C'est ce prédicat qui fait le lien entre les prédicats de choix de transformation (*regle*) et ceux d'application de cette transformation.

```
unifie([E|P]) :-
write("system: "),print([E|P]),nl,
regle(E, R),
write(R), write(": "), write(E), nl,
reduit(R, E, P, Q),
unifie(Q)
.
```

Listing 3 – Prédicat d'unification d'un système d'équation

Question 2

La rapidité d'exécution de l'algorithme dépend du choix plus ou moins judicieux des règles à exécuter en premier sur le système d'équation. C'est pourquoi il faut mettre en place un système qui prenne en compte le l'ordre dans lequel les équations (ou les règles) sont choisies. Nous avons donc décidé d'implémenter quatre méthodes de choix :

- Choix du premier : la première équation de l'ensemble est résolue
- Choix pondéré : les équations sont choisies en fonction des règles que l'on peut appliquer dessus. Les règles sont classées dans un ordre préétablit qui est plus judicieux dans certains cas . Pour notre exemple les règles *clash* et *check* sont prioritaire, ce qui permet d'arrêter l'exécution de l'algorithme au plus tôt si jamais le système d'équation est insolvable.
- Choix aléatoire : les équations sont prise de façon aléatoire, on y applique la règle qui est possible
- Choix du dernier : la dernière équation du système est choisie

Grâce à ces quatre possibilités, nous pouvons voir la différence d'exécution entre ces différentes méthodes de choix d'équation :

```
3 ?- trace_unif([f(X, Y, h(C, V)) ?= f(g(Z), h(a), h(e, r)), Z ?= f(Y), f(g) ?= f(a, b)], premier).
system: [f(_G1995,_G1996,h(_G1992,_G1993))?=f(g(_G1999),h(a),h(e,r)),_G1999?=f(_G1996),f(g)?=f(a,b)]
decompose: f(_G1995,_G1996,h(_G1992,_G1993))?=f(g(_G1999),h(a),h(e,r))
system: [_G1995?=g(_G1999),_G1996?=h(a),h(_G1992,_G1993)?=h(e,r),_G1999?=f(_G1996),f(g)?=f(a,b)]
expand: _G1995?=g(_G1999)
system: [_G1996?=h(a),h(_G1992,_G1993)?=h(e,r),_G1999?=f(_G1996),f(g)?=f(a,b)]
expand: _G1996?=h(a)
system: [h(_G1992,_G1993)?=h(e,r),_G1999?=f(h(a)),f(g)?=f(a,b)]
decompose: h(_G1992,_G1993)?=h(e,r)
system: [_G1992?=e,_G1993?=r,_G1999?=f(h(a)),f(g)?=f(a,b)]
simplify: _G1992?=e
system: [_G1993?=r,_G1999?=f(h(a)),f(g)?=f(a,b)]
simplify: _G1993?=r
system: [_G1999?=f(h(a)),f(g)?=f(a,b)]
expand: _G1999?=f(h(a))
system: [f(g)?=f(a,b)]
clash: f(g)?=f(a,b)
No
```

Listing 4 – Exemple d'exécution avec le choix de la première équation

```
4 ?- trace_unif([f(X, Y, h(C, V)) ?= f(g(Z), h(a), h(e, r)), Z ?= f(Y), f(g) ?= f(a, b)], pondere).
system: [f(_G1995,_G1996,h(_G1992,_G1993))?=f(g(_G1999),h(a),h(e,r)),_G1999?=f(_G1996),f(g)?=f(a,b)]
clash: f(g)?=f(a,b)
No
```

Listing 5 – Exemple d'exécution avec le choix pondéré

```
5 ?- trace_unif([f(X, Y, h(C, V)) ?= f(g(Z), h(a), h(e, r)), Z ?= f(Y), f(g) ?= f(a, b)], aleatoire).
system: [f(_G2007,_G2008,h(_G2004,_G2005))?=f(g(_G2011),h(a),h(e,r)),_G2011?=f(_G2008),f(g)?=f(a,b)]
expand: _G2011?=f(_G2008)
system: [f(_G2007,_G2008,h(_G2004,_G2005))?=f(g(f(_G2008)),h(a),h(e,r)),f(g)?=f(a,b)]
clash: f(g)?=f(a,b)
No
```

Listing 6 – Exemple d'exécution avec le choix aléatoire

On constate donc que sur ce cas particulier, le choix pondéré a bien fonctionné, c'est à dire que l'algorithme a tout de suite repéré que le système était insolvable et a stoppé l'exécution, le choix aléatoire s'en sort bien lui aussi mais ça aurait très bien pu ne pas être le cas, le choix de la première équation est le pire des cas puisqu'on ne tombe sur le problème qu'une fois toutes les autres équations résolues.

Question 3

Pour passer du fonctionnement de la question 2 à celui de la question 3. C'est-à-dire créer les prédicats $unif(P, S)$ et $trace_unif(P, S)$ permettant respectivement d'exécuter unifie sans et avec les messages dans la console. Il a suffit de remplacer tous les appels aux fonctions "write" et "print" par un appel à la fonction "echo", les nouveaux prédicats font tout deux appels à unifie, mais ils précèdent cet appel par clr_echo ou set_echo, servant respectivement à désactiver ou activer le flag echo_on. Ainsi quand le prédicat unifie appellera "echo" ce dernier affichera ou n'affichera pas le message en fonction de l'état du flag mais continuera son exécution quoi qu'il arrive.

```
?- trace_unif([a ?= U, f(X, Y) ?= f(g(Z ),h(a )), Z ?= f(Y )], premier).
system: [a?=_G192,f(_G197,_G198)?=f(g(_G200),h(a )),_G200?=f(_G198)]
orient: a?=_G192
system: [_G192?=a,f(_G197,_G198)?=f(g(_G200),h(a )),_G200?=f(_G198)]
simplify: _G192?=a
system: [f(_G197,_G198)?=f(g(_G200),h(a )),_G200?=f(_G198)]
decompose: f(_G197,_G198)?=f(g(_G200),h(a ))
system: [_G197?=g(_G200),_G198?=h(a),_G200?=f(_G198)]
expand: _G197?=g(_G200)
system: [_G198?=h(a ),_G200?=f(_G198)]
expand: _G198?=h(a )
system: [_G200?=f(h(a ))]
expand: _G200?=f(h(a ))
Yes
U = a,
X = g(f(h(a ))),
Y = h(a ),
Z = f(h(a )).
```

Listing 7 – Exemple d'execution avec un niveau de debug

```
unif([a ?= U, f(X, Y) ?= f(g(Z),h(a)), Z ?= f(Y)], premier).
U = a,
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)).
```

Listing 8 – Exemple d'exécution

Annexe A

Sources

Les sources de ce projet sont divisées en 2 fichiers. Le fichier principal est le fichier *projet.pl*, il contient le programme demandé. Le second fichier est le fichier *test.pl* qui contient les tests unitaires que nous avons créés pour s'assurer que notre programme faisait bien le travail voulu, notamment les prédicats *regle* et *reduit*. Les tests se lance avec le prédicat *tests.*. Les sources sont aussi consultables en ligne à l'adresse `www.github.com/lesquoyb/martelli_montanari` .

A.1 projet

```
1  :- op(20,xfy,?=).
2
3  % Predicats d'affichage fournis
4
5  % set_echo: ce predicat active l'affichage par le predicat echo
6  set_echo :- assert(echo_on).
7
8  % clr_echo: ce predicat inhibe l'affichage par le predicat echo
9  clr_echo :- retractall(echo_on).
10
11 % echo(T): si le flag echo_on est positionne, echo(T) affiche le terme T
12 %          sinon, echo(T) reussit simplement en ne faisant rien.
13
14 echo(T) :- echo_on, !, write(T).
15 echo(_).
16
17 %select_strat choisi une equation a l'aide de predicat correspondant a la strategie passee en parametre
18 select_strat(premier, P, E, R):-
19   choix_premier(P, _, E, R)
20 .
21
22 select_strat(pondere, P, E, R):-
23   choix_pondere(P, _, E, R)
24 .
25
26 select_strat(aleatoire, P, E, R):-
27   choix_aleatoire(P, _, E, R)
28 .
29 select_strat(dernier, P, E, R):-
30   choix_dernier(P, _, E, R)
31 .
32
33 %le coeur du programme, essaye d'unifier P en utilisant la strategie Strat, si on ne passe pas de strategie il
       choisira la strategie "premiere equation"
34 unifie(P):- unifie(P, premier).
35 unifie([], _):- true, !.
36 unifie(bottom, _):- false, !. %Si on a bottom => c'est un echec
37 unifie(P, Strat):-
38   echo("system: "),echo(P),echo("\n"),
39   select_strat(Strat, P, E, R),
```

```
  echo(R),echo(": "),echo(E),echo("\n"),
  reduit(R, E, P, Q),
  unifie(Q, Strat), !
.
%appelle unifie apres avoir desactive les affichages
unif(P, S):-
  clr_echo,
  unifie(P, S)
.
%appelle unifie apres avoir active les affichages, affiche "Yes" si on peut unifier "No" sinon (il n'y a donc pas d'echec de la procedure.
trace_unif(P,S):-
  set_echo,
  (
    unifie(P, S), echo("Yes"),!
  ;
    echo("No")
  )
.
%sous fonction du predicat de selection d'equation avec choix pondere
select_rule([], _, _, _):- false, !.%on a parcouru la liste des regle sans en trouver une qui fonctionne
select_rule( [Next | MasterList], [], Pbase, P, R, E):-%on a vide le sous groupe de regle qu'on etait en train de traiter, on passe au suivant
  select_rule(MasterList, Next, Pbase, P, R, E)
.
select_rule(MasterList, [ _ | ListRules], Pbase, [], R, E):-%on a parcourus toutes les equations sans pouvoir appliquer la regle voulue, on passe a la regle suivante, et on reinitialise les equations
  select_rule(MasterList, ListRules, Pbase, Pbase, R, E)
.
select_rule( MasterList,[FirstRule | ListRules], Pbase, [Ep|P], R, E):-%on essaye d'appliquer la regle FirstRule a Ep, si on echoue, on recommence avec l'equation suivante
  (
    regle(Ep, FirstRule),
    R = FirstRule,
    E = Ep,
    !
  ;
    select_rule(MasterList, [FirstRule | ListRules], Pbase, P, R, E),!
  )
.
%remplie la liste pondere de regle, il s'agit d'une liste de liste. Chaque sous liste represente un groupe de regles de meme importance (oui on aurait pu faire une simple liste, mais c'est surement plus evolutif comme ca )
liste_pondere([ [clash, check],
        [rename, simplify],
        [orient, decompose],
        [expand]
        ]):-
  true
.
%choix de la premiere equation dans P
choix_premier( [E|_], _, E, R):-
  regle(E,R),!
.
%choix de la premiere equation satisfaisant la regle de plus haute importance
choix_pondere(P, _, E, R):-
  liste_pondere( [FirstRules | List] ),
  select_rule(List,FirstRules, P, P, R, E)
.
%choix d'une equation aleatoire
choix_aleatoire(P, _, E, R):-
  random_member(E, P),
  regle(E, R),
  !
.
%choix de la derniere equation dans P
```

```
choix_dernier(P, _, E, R):-
  reverse(P, [E|_]),
  regle(E, R),
  !
.
%regle teste si on peut appliquer la regle R (deuxieme parametre) a l'equation E (premier parametre)
regle(E, decompose):-
  not(atom(E)),
  split(E, X, Y),
  compound(X),
  compound(Y),
  compound_name_arity(X, N, A),
  compound_name_arity(Y, N, A)
.
regle(E, simplify):-
  split(E, L, R),
  var(L),
  not(var(R)),
  not(compound(R))
.
regle(E, rename):-
  split(E, L, R),
  var(R),
  var(L)
.
regle(E, expand):-
  split(E, X, Y),
  var(X),
  compound(Y),
  not(occur_check(X, Y))
.
regle(E, clash):-
  split(E, L, R),
  compound(L),
  compound(R),
  compound_name_arity(L, Nl, Al),
  compound_name_arity(R, Nr, Ar),
  not( (Nl == Nr, Al == Ar) )
.
regle(E, check):-
  split(E, X, Y),
  not(X == Y),
  var(X),
  occur_check(X, Y)
.
regle(E, orient):-
  split(E, L, R),
  var(R),
  not(var(L))
.

%Retourne L et R les arguments 1 et 2 de E ( utilise pour couper une expression de type "X ?= Y" en deux)
split(E, L, R):-
  arg(1, E, L),
  arg(2, E, R)
.
%est vrai si V est une variable qui apparait dans T
occur_check(V, T) :-
  var(V),
  compound(T),
  term_variables(T, L),
  occur_check_list(V, L)
.
%sous fonction qui verifie parametre par parametre de T si V apparait
occur_check_list(_, []):-%on a parcouru tous les parametres sans rien trouver
  not(true)%parce que pourquoi avoir une constante false quand on peut faire not(true)
```

```
.
occur_check_list(V, [C|T]) :-%recursivite pour avancer dans la liste
  occur_check_list(V, T);
  V == C
.
%transforme f(x1,x2 ..., xn) ?= f(y1,y2 ,..., yn) en [x1 ?= y1, x2 ?= y2, ... , xn ?= yn]
unif_list([], [],[]):- true.
unif_list([L|List1], [R|List2], [L ?= R| Rp]):-
  unif_list(List1, List2, Rp)
.
%reduit applique la regle R (premier argument) a l'equation E appartenant a P, et renvoie le nouvel ensemble Q
reduit(simplify, E, P, Q) :-
  split(E, X, T),
  X = T,
  delete_elem(E, P, Q)
.
reduit(rename, E, P, Q):-
  split(E, X, T),
  X = T,
  delete_elem(E, P, Q)
.
reduit(expand, E, P, Q) :-
  split(E, X, T),
  X = T,
  delete_elem(E, P, Q)
.
reduit(check, _, _, bottom):- false .

reduit(orient, E, P, [ R ?= L | Tp ]) :-
  split(E, L, R),
  delete_elem(E, P, Tp)
.
reduit(decompose, E, P, S):-
  split(E, L, R),
  L =.. [_|ArgsL],
  R =.. [_|ArgsR],
  unif_list(ArgsL, ArgsR, Res),
  delete_elem(E, P, Pp),
  union(Res, Pp, S)
.
reduit(clash, _, _, bottom):- false .
%supprime l'element Elem de la list en deuxieme parametre et renvoie Set le nouvel ensemble
delete_elem(_, [], []) :- !.
delete_elem(Elem, [Elem|Set], Set):- ! .
delete_elem(Elem, [E|Set], [E|R]):-
  delete_elem(Elem, Set, R)
.
```

Listing A.1 – le fichier projet.pl

A.2 tests

```
:-
  op(20, xfy,[?=]),
  [projet]
.

writeOK() :- write(" : ok"), nl.

tests() :-
  writeln("Debut des tests : "), nl,
  writeln("==== Test : Reduit ===="), nl,
  test_reduit_decompose,
  test_reduit_rename,
  test_reduit_simplify,
  test_reduit_expand,
```

```
  test_reduit_orient,
  test_reduit_clash,
  test_reduit_check,
  writeln("Reduit : checked"),

  nl, writeln("==== Test : Regle ===="), nl,
  test_regle_simplify,
  test_regle_rename,
  test_regle_expand,
  test_regle_orient,
  test_regle_decompose,
  test_regle_clash,
  test_regle_check,
  writeln("Regle : checked"),
  /*
  On a supprime car le fonctionnnement de unifie a change au cours du TP, et ce n'est pas necessaire que ce soit
      teste car les fonctions qui constituent unifie sont testees independemment.
  nl, writeln("======= Test: Unifie ====="),nl,
  test_unifie(),
  */
  write("Tous les tests sont passe avec succes")
.

test_unifie():-
  unifie([f(X, Y) ?= f(g(Z), h(a)), Z ?= f(Y)]),
  write("exemple prof juste: ok"),nl,nl,
  not(unifie([f(X, Y) ?= f(g(Z), h(a)), Z ?= f(X)])),
  write("exemple prof faux: ok"),nl,nl,
  not(unifie([a?=b])),
  not(unifie([a ?= X, X ?= b]))
.

test_reduit_decompose():-
  writeln("==== Reduit : Decompose ===="),

  write("f(a) ?= f(b), [a ?= b]"),
  reduit(decompose, f(a) ?= f(b), [f(a)?=f(b)], [a?=b]),
  writeOK,

  write("f(a) ?= f(a), [a ?= a]"),
  reduit(decompose, f(a) ?= f(a), [], [a?=a]),
  writeOK,

  write("f(a,b,c) ?= f(d, e, f), [a ?= d, b ?= e, c ?= f]"),
  reduit(decompose, f(a, b, c) ?= f(d, e, f), [], [a?=d, b ?= e, c ?= f ]),
  writeOK,


  write("f(X) ?= f(a), [X ?= a]"),
  reduit(decompose, f(X) ?= f(a), [], [X?=a]),
  writeOK,


  write("f(a, X , b, Y) ?= f(d, e, f, g), [a ?= d, X ?= e, b ?= f, Y?= g]"),
  reduit(decompose, f(a, X, b, Y) ?= f(d, e, f, g), [], [a?=d, X ?= e, b ?= f, Y ?= g]),
  writeOK,


write("f(g(X), W) ?= f(A, Q), [g(X) ?= A, W ?= Q]"),
  reduit(decompose, f(g(X), W) ?= f(A, Q), [f(g(X), W) ?= f(A, Q)], [g(X)?=A,W?=Q]),
  writeOK
.

test_reduit_rename() :-
  writeln("==== Reduit : Rename ===="),
  write("X?=Y, []"),
```

```
  reduit(rename, X2?=Y2, [X2?=Y2], []),
  X2 == Y2,
  writeOK,

  write("X?=Y,[X ?= a] => [Y ?= a]"),
  reduit(rename, X1?=Y1, [X1?=Y1, X1 ?= a], [Y1 ?= a]),
  X1 == Y1,
  writeOK

.

test_reduit_simplify() :-
  writeln("==== Reduit : Simplify ===="),

  write("X?=a , []"),
  reduit(simplify, X?=a, [X?=a], []),
  writeOK,
  write("X?=a,[Y ?= X] => [Y ?= a]"),
  reduit(simplify, X1?=a, [Y1?=X1, Y1 ?= X1], [Y1 ?= a]),
  writeOK
.

test_reduit_expand() :-
  writeln("==== Reduit : Expand ===="),

  write("X?=f(a), []"),
  reduit(expand, X2?=f(a), [X2?=f(a)], []),
  X2 == f(a),
  writeOK,

  write("X?=f(E), []"),
  reduit(expand, X3?=f(E2), [X3?=f(E2)], []),
  X3 == f(E2),
  writeOK
.

test_reduit_orient() :-
  writeln("==== Reduit : Orient ===="),

  write("f(W)?=X, [X?=f(W)]"),
  reduit(orient, f(W)?=X, [f(W)?=X], [X?=f(W)]),
  writeOK,

  write("f(a)?=X, [X?=f(a)]"),
  reduit(orient, f(a)?=X, [f(a)?=X], [X?=f(a)]),
  writeOK
.

test_reduit_clash() :-
  writeln("==== Reduit : clash ===="),

  write("clash"),
  not(reduit(clash, _, _, bottom)),
  writeOK
.

test_reduit_check() :-
  writeln("==== Reduit : check ===="),

  write("check"),
  not(reduit(check, _, _, bottom)),
  writeOK
.

test_regle_simplify() :-
  writeln("==== Regle : simplify ===="),
```

```
146
147   write("X ?= a"),
148   regle(X?=a, simplify),
149   writeOK,
150
151   write("X ?= 1"),
152   regle(X44 ?= 1, simplify),
153   writeOK,
154
155
156   write("a ?= b"),
157   not(regle(a ?= b, simplify)),
158   writeOK,
159
160   write("X ?= f(a)"),
161   not(regle(X1 ?= f(a), simplify)),
162   writeOK,
163
164   write("X ?= Y"),
165   not(regle(X2 ?= Y, simplify)),
166   writeOK,
167
168
169   write("X ?= f(Y)"),
170   not(regle(X3 ?= f(Y), simplify)),
171   writeOK,
172
173   write("f(a) ?= W"),
174   not(regle(f(a) ?= W1, simplify)),
175   writeOK,
176
177   write("f(a) ?= f(W)"),
178   not(regle(f(a) ?= f(W2), simplify)),
179   writeOK
180 .
181
182 test_regle_rename() :-
183   writeln("==== Regle : rename ===="),
184
185   write("X ?= a"),
186   not(regle(X1?=a, rename)),
187   writeOK,
188
189   write("X ?= f(a)"),
190   not(regle(X2 ?= f(a), rename)),
191   writeOK,
192
193   write("X ?= Q"),
194   regle(X3?=Q1, rename),
195   writeOK,
196
197   write("a ?= W"),
198   not(regle(a ?= W1, rename)),
199   writeOK,
200
201   write("a ?= b"),
202   not(regle(a ?= b, rename)),
203   writeOK,
204
205   write("a ?= f(b)"),
206   not(regle( a ?= f(b), rename)),
207   writeOK,
208
209
210   write("f(a) ?= W"),
211   not(regle(f(a) ?= W2, rename)),
```

```
  writeOK,

  write("f(a) ?= f(W)"),
  not(regle(f(a) ?= f(W3), rename)),
  writeOK
.

test_regle_expand() :-
  writeln("==== Regle : expand ===="),

  write("X ?= a"),
  not(regle(X1 ?= a, expand)),
  writeOK,

  write("X ?= f(a)"),
  regle(X2 ?= f(a), expand),
  writeOK,

  write("X ?= f(Q)"),
  regle(X3 ?= f(Q1), expand),
  writeOK,


  write("X ?= f(X)"),
  not(regle(X10 ?= f(X10), expand)),
  writeOK,


  write("X ?= f(a, b, X)"),
  not(regle(X12 ?= f(a, b, X12), expand)),
  writeOK,

  write("X ?= f(a, b, c, g(X))"),
  not(regle(X11 ?= f(a, b, c,g(X11) ), expand)),
  writeOK,

  write("X ?= Q"),
  not(regle( X4 ?=Q2, expand)),
  writeOK,

  write("f(a) ?= W"),
  not(regle(f(a) ?= W1, expand)),
  writeOK,

  write("f(a) ?= f(W)"),
  not(regle(f(a) ?= f(W2), rename)),
  writeOK
.

test_regle_orient() :-
  writeln("==== Regle : orient ===="),

  write("X ?= a"),
  not(regle(X1?=a, orient)),
  writeOK,

  write("X ?= f(a)"),
  not(regle(X2 ?= f(a), orient)),
  writeOK,

  write("X ?= Q"),
  not(regle(X3 ?=Q, orient)),
  writeOK,

  write("a ?= W"),
  regle(a ?= W1, orient),
```

```
278   writeOK,
279
280   write("f(a) ?= W"),
281   regle(f(a) ?= W2, orient),
282   writeOK,
283
284   write("f(X) ?= W"),
285   regle(f(X) ?= W4, orient),
286   writeOK,
287
288   write("f(a) ?= f(W)"),
289   not(regle(f(a) ?= f(W3), orient)),
290   writeOK
291 .
292 test_regle_decompose() :-
293   writeln("==== Regle : decompose ===="),
294
295   write("X ?= a"),
296   not(regle( X ?= a, decompose)),
297   writeOK,
298
299   write("X ?= f(a)"),
300   not(regle(X ?= f(a), decompose)),
301   writeOK,
302
303   write("X ?= Q"),
304   not(regle(X?=Q, decompose)),
305   writeOK,
306
307   write("f(X) ?= W"),
308   not(regle(f(X) ?= W, decompose)),
309   writeOK,
310
311   write("f(a) ?= g(a)"),
312   not(regle(f(a) ?= g(a), decompose)),
313   writeOK,
314
315   write("f(a) ?= f(a,b)"),
316   not(regle(f(a) ?= f(a, b), decompose)),
317   writeOK,
318
319   write("f(a,b) ?= f(a)"),
320   not(regle(f(a,b) ?= f(a), decompose)),
321   writeOK,
322
323   write("f(a,g()) ?= f(b, c())"),
324   regle(f(a, g()) ?= f(b, c()), decompose),
325   writeOK,
326
327
328   write("f(a, X) ?= f(a, b)"),
329   regle(f(a, X) ?= f(a,b), decompose),
330   writeOK,
331
332   write("f(a) ?= f(W)"),
333   regle(f(a) ?= f(W), decompose),
334   writeOK,
335
336   write("f(a, E, q) ?= f(W, c, e)"),
337   regle(f(a) ?= f(W), decompose),
338   writeOK
339 .
340
341 test_regle_clash() :-
342   writeln("==== Regle : clash ===="),
343
```

```
  write("X ?= a"),
  not(regle(X?=a, clash)),
  writeOK,

  write("X ?= f(a)"),
  not(regle(X ?= f(a), clash)),
  writeOK,

  write("X ?= Q"),
  not(regle(X?=Q, clash)),
  writeOK,

  write("f(X) ?= W"),
  not(regle(f(X) ?= W, clash)),
  writeOK,

  write("f(a) ?= f(W)"),
  not(regle(f(a) ?= f(W), clash)),
  writeOK,

  write("f(Q, E) ?= g(E, V)"),
  regle(f(Q, E) ?= g(E, V), clash),
  writeOK,

  write("f(Q) ?= f(E, V)"),
  regle(f(Q, E) ?= g(E, V), clash),
  writeOK,

  write("f(x, y) ?= g(x, y)"),
  regle(f(x, y) ?= g(x, y), clash),
  writeOK,

  write("f(Q, E) ?= g(E, V, e)"),
  regle(f(Q, E) ?= g(E, V), clash),
  writeOK
.
test_regle_check() :-
  writeln("==== Regle : check ===="),

  write("X ?= a"),
  not(regle(X?=a, check)),
  writeOK,

  write("X ?= f(a)"),
  not(regle(X ?= f(a), check)),
  writeOK,

  write("X ?= Q"),
  not(regle(X?=Q, check)),
  writeOK,

  write("f(X) ?= W"),
  not(regle(f(X) ?= W, check)),
  writeOK,

  write("f(a) ?= f(W)"),
  not(regle(f(a) ?= f(W), check)),
  writeOK,

  write("f(Q, E) ?= g(E, V)"),
  not(regle(f(Q, E) ?= g(E, V), check)),
  writeOK,

  write("X ?= f(g(X))"),
  regle(X ?= f(g(X)), check),
  writeOK,
```

```
410
411   write("X ?= f(X)"),
412   regle(X ?= f(X), check),
413   writeOK,
414
415   write("X ?= f(a,X)"),
416   regle(X ?= f(a,X), check),
417   writeOK
418 .
```

Listing A.2 – le fichier test.pl