Rapport de projet de Logique et Modèles de Calcul

Algorithme d'unification Martelli-Montanari

GROSJEAN Lucas
PHEULPIN Naomi

2022-2023

1. Règles de transformation

Les règles qui suivent permettent de savoir si ladite règle est applicable ou non sur une équation donnée.

1.1. Règle Rename

La règle de renommage de variable prend en entrée une équation du type X?=T et vérifie si les deux arguments X et T sont des variables. Si c'est le cas, le prédicat réussit.
Cette règle est définie comme suit :

```
regles(X?=T, rename) :- var(X), var(T), !.
```

1.2. Règle Simplify

La règle de simplification d'une constante prend en entrée une équation du type X?=T et vérifie si X est une variable et T une constante. Si c'est le cas, le prédicat réussit.
Cette règle est définie comme suit :

```
regles(X?=T, simplify) :- var(X), atomic(T), !.
```

1.3. Règle Expand

La règle de développement prend en entrée une équation du type X?=T et vérifie si X est une variable et T un terme composé et que X n'apparaît pas dans T. Si c'est le cas, le prédicat réussit.
Cette règle est définie comme suit :

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

1.4. Règle Check

La règle de d'occurrence prend en entrée une équation du type X?=T et vérifie si X est une variable et T un terme différent de X et que X n'apparaît pas dans T. Si c'est le cas, le prédicat réussit.
Cette règle est définie comme suit :

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

1.5. Règle Orient

La règle de d'orientation prend en entrée une équation du type T?=X et vérifie si X est une variable et T un terme. Si c'est le cas, le prédicat réussit.
Cette règle est définie comme suit :

```
regles(T?=X, orient) :- var(X), nonvar(T), !.
```

1.6. Règle Decompose

La règle de décomposition prend en entrée une équation du type F?=G et vérifie si F et G sont des termes composés ainsi que F et G portent le même nom et possède le même nombre d'arguments. Si c'est le cas, le prédicat réussit.
Cette règle est définie comme suit :

```
regles(F?=G, decompose) :- compound(F), compound(G),
                           functor(F, NAME, ARITY),
                           functor(G, NAME, ARITY), !.
```

1.7. Règle Clash

La règle de gestion de conflit prend en entrée une équation du type F?=G et vérifie si F et G sont des termes composés. Mais aussi que F et G ne portent pas le même nom ou ne possède pas le même nombre d'arguments. Si c'est le cas, le prédicat réussit.
Cette règle est définie comme suit :

```
regles(F?=G, clash) :- compound(F), compound(G),
        functor(F, FNAME, FARITY), functor(G,GNAME, GARITY),
        (FNAME \== GNAME ; FARITY \== GARITY), !.
```

2. Prédicat d'occurence occur_check

Le prédicat d'occurrence prend en entrée deux termes et vérifie si V est une variable, T un terme composé et si V apparaît dans T. Si c'est le cas, le prédicat réussit.
Ce prédicat est défini comme suit :

```
occur_check(V, T) :- var(V), compound(T), contains_var(V, T), !.
```

3. Prédicats de réduction (reduit)

Un prédicat de réduction prend quatre paramètres :

- une règle : R
- une équation : E
- un système d'équation : P
- une variable qui correspond au résultat de l'application de la règle sur l'équation : Q

Un prédicat de réduction 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.

3.1. Rename

Le prédicat *rename* renomme toutes les occurrences de X par T dans le système d'équation P et enregistre le résultat dans Q.

```
reduit(rename, X?=T, P, Q) :- X=T, Q=P.
```

3.2. Simplify

Le prédicat *simplify* renomme toutes les occurrences de X par T dans le système d'équation P et enregistre le résultat dans Q.

```
reduit(simplify, X?=T, P,Q) :- X=T, Q=P.
```

3.3. Expand

Le prédicat *expand* remplace toutes les occurrences de X par T dans le système d'équation P et enregistre le résultat dans Q.

```
reduit(expand, X?=T, P, Q) :- X=T, Q=P.
```

3.4. Check

Le prédicat *check* échoue, car un *occur check* a été trouvé, cela arrête l'exécution des règles d'unification.

```
reduit(check, _, _, _):- echo("\n No\n"), fail.
```

3.5. Orient

Le prédicat *orient* permute le terme T et la variable X dans l'équation donnée et enregistre le nouveau système d'équation dans Q.

```
reduit(orient, T?=X, P, Q) :- append([X?=T], P, Q).
```

3.6. Decompose

Le prédicat *decompose* met en relation deux à deux les arguments des termes composés F et G sous l'opérateur *?*= et enregistre le nouveau système d'équation dans Q.

```
reduit(decompose, F?=G, P, Q) :-
        F =.. [_|FARGS], G =.. [_|GARGS],
        decomposition(FARGS, GARGS, RES),
        append(RES, P, Q).
```

3.7. Clash

Le prédicat *clash* échoue, car un *clash* a été trouvé, cela arrête l'exécution des règles d'unification.

```
reduit(clash, _, _, _):- echo("\n No\n"), fail.
```

4. Prédicat decomposition

Le prédicat de décomposition prend en entrée deux listes de termes et une variable RES où sera enregistrer la liste des équations résultante. On prend le premier élément des deux listes et on les met en relation sous l'opérateur *?*=, on applique cela récursivement sur l'ensemble de la liste. Si les listes d'entrée sont vides, cela arrête la récursion.
Ce prédicat est défini comme suit :

```
decomposition([H1|T1], [H2|T2], RES) :-
       decomposition(T1, T2, ACC),append([H1?=H2], ACC, RES).
decomposition([], [], []).
```

5. Prédicats « outils »

Le prédicat *main* sert d'aide à l'utilisateur pour l'exécution du programme, il active par défaut l'affichage des étapes d'unification.

5.1. Les prédicats d'affichage

5.1.1. Prédicat d'affichage système aff_sys

Le prédicat d'affichage de l'état du système prend en entrée une liste d'équation et l'affiche à l'utilisateur.

```
aff_sys(P) :- echo('system: '),echo(P),echo('\n').
```

5.1.2. Prédicat d'affichage règle aff_règle

Le prédicat d'affichage d'une règle prend en entrée une équation et une règle puis l'affiche à l'utilisateur.

```
aff_regle(E,R) :- echo(R),echo(': '),echo(E),echo('\n').
```

5.1.3. Prédicat d'unification sans traces

Le prédicat d'unification sans traces prend en entrée une liste d'équations et une stratégie à appliquer. Il désactive l'affichage des étapes puis unifie selon la stratégie saisie.

```
unif(P,S) :- clr_echo, unifie(P,S).
```

5.1.4. Prédicat d'unification avec traces

Le prédicat d'unification avec traces prend en entrée une liste d'équations et une stratégie à appliquer. Il active l'affichage des étapes puis unifie selon la stratégie saisie.

```
trace_unif(P,S) :- set_echo, unifie(P,S).
```

5.2. Les prédicats pre-unification

5.2.1. Prédicat d'association de poids

Le prédicat d'association de poids prend en entrée une équation et une variable pour enregistrer le résultat. Il vérifie si des règles sont applicables sur l'équation par ordre décroissant d'importance et renvois le poids de la première règle applicable.
clash, check > rename, simplify > orient > decompose > expand.

```
get_i(E, I) :- (regles(E, clash); regles(E, check)) -> I = 5, !;
               (regles(E, rename); regles(E, simplify)) -> I = 4, !;
                 regles(E, orient) -> I = 3, !;
                 regles(E, decompose) -> I = 2, !;
                 regles(E, expand) -> I = 1, !.
```

5.2.2. Prédicat sélection d'équation pondéré

Le prédicat de sélection d'équation pondéré prend en entrée une liste d'équations, une variable servant d'accumulateur aux appels récursif, une équation, et une variable stockant le résultat. Il compare à chaque appel le premier élément de la liste avec l'équation d'entrée. Si son poids est plus faible ou égal, on l'ajoute au début de l'accumulateur et on rappelle la fonction. Sinon, le premier élément de la liste prend la place de l'équation d'entrée et on ajoute l'ancienne au début de l'accumulateur. Si la liste d'équations est vide, l'équation d'entrée est donc le poids le plus élevé, on affecte donc à la variable RES l'accumulateur avec l'équation au début.

```
choix_eq([H|T], TMP, E, RES) :- get_i(E, I1), get_i(H, I2),
                                (I1>=I2 -> append([H], TMP, NEWTMP),
                                           choix_eq(T, NEWTMP, E, RES);
                                          (choix_eq(T, [E|TMP], H, RES))
                                ).
choix_eq([], TMP, E, RES) :- append([E], TMP , RES).
```

5.2.3. Prédicat sélection dernière équation

Le prédicat de sélection de dernière équation prend en entrée une liste d'équations et une liste stockant le résulat. Il récupère le dernier élément de la liste d'entrée et le place au début.

```
last_list(P, [LAST|RESTE]) :- reverse(P, [LAST|R]), reverse(R, RESTE).
```

6. Prédicat d'unification unifie_

Le prédicat d'unification "général" prend en entrée une liste d'équations et la stratégie à utiliser. Elle affiche l'état du système, puis trouve la règle à utiliser sur le premier élément de la liste d'équation. Une fois trouvé, affiche la règle utilisée sur cet élément puis utilise le prédicat de réduction adapté. Enfin, *unifie_* appel le prédicat appelant. L'unification se termine lorsqu'il est appelé avec la constante "*fin*", cela signifiant que l'unification se termine sur un succès.

```
unifie_([H|T], S) :- aff_sys(H|T), regles(H, R),
                     aff_regle(H, R), reduit(R, H, T, Q),
                     unifie(Q, S).
unifie_(fin) :- echo("\n Yes\n").
```

7. Stratégies d'unification

7.1. Stratégie choix_premier

Le prédicat d'unification choix_premier prend en entrée une liste d'équation à unifier et la constante choix_premier. Il se contente d'appeler le prédicat d'unification "général" *unifie_* avec ses arguments d'entrée.

```
unifie(P, choix_premier) :- unifie_(P, choix_premier), !.
```

7.2. Stratégie choix_pondere

Le prédicat d'unification choix_pondere prend en entrée une liste d'équation à unifier et la constante choix_pondere. Il fait appel au prédicat de sélection d'équation pondéré et injecte le résultat en argument de l'appel du prédicat d'unification "général" *unifie_* avec la constante choix_pondere.

```
unifie([H|T], choix_pondere) :- choix_eq(T, [], H, Q), unifie_(Q,
                        choix_pondere), !.
```

7.3. Stratégie choix_dernier

Le prédicat d'unification choix_dernier prend en entrée une liste d'équation à unifier et la constante choix_dernier. Il fait appel au prédicat de sélection de dernière équation et injecte le résulat en argument de l'appel du prédicat d'unification "général" *unifie_* avec la constante choix_dernier

```
unifie(P, choix_dernier) :- last_list(P, RES), unifie_(RES,
                            choix_dernier), !.
```

L'unification se termine lorsque la liste d'équation est vide, peu importe la stratégie, et fait appel au prédicat d'unification "général" *unifie_* avec la constante "*fin*".

8. Exécution du programme

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_premier).
system: f(_3418,_3420)?=f(g(_3424),h(a))|[_3424?=f(_3420)]
decompose: f(_3418,_3420)?=f(g(_3424),h(a))
system: _3418?=g(_3424)|[_3420?=h(a),_3424?=f(_3420)]
expand: _3418?=g(_3424)
system: _3420?=h(a)|[_3424?=f(_3420)]
expand: _3420?=h(a)
system: _3424?=f(h(a))|[]
expand: _3424?=f(h(a))

 Yes
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)) .
```

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_pondere).
system: f(_12698,_12700)?=f(g(_12704),h(a))|[_12704?=f(_12700)]
decompose: f(_12698,_12700)?=f(g(_12704),h(a))
system: _12698?=g(_12704)|[_12704?=f(_12700),_12700?=h(a)]
expand: _12698?=g(_12704)
system: _12704?=f(_12700)|[_12700?=h(a)]
expand: _12704?=f(_12700)
system: _12700?=h(a)|[]
expand: _12700?=h(a)

 Yes
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)).
```

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)], choix_dernier).
system: _17000?=f(_16996)|[f(_16994,_16996)?=f(g(_17000),h(a))]
expand: _17000?=f(_16996)
system: f(_16994,_16996)?=f(g(f(_16996)),h(a))|[]
decompose: f(_16994,_16996)?=f(g(f(_16996)),h(a))
system: _16996?=h(a)|[_16994?=g(f(_16996))]
expand: _16996?=h(a)
system: _16994?=g(f(h(a)))|[]
expand: _16994?=g(f(h(a)))

 Yes
X = g(f(h(a))),
Y = h(a),
Z = f(h(a)).
```

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)], choix_premier).
system: f(_20030,_20032)?=f(g(_20036),h(a))|[_20036?=f(_20030)]
decompose: f(_20030,_20032)?=f(g(_20036),h(a))
system: _20030?=g(_20036)|[_20032?=h(a),_20036?=f(_20030)]
expand: _20030?=g(_20036)
system: _20032?=h(a)|[_20036?=f(g(_20036))]
expand: _20032?=h(a)
system: _20036?=f(g(_20036))|[]
check: _20036?=f(g(_20036))

 No
false.
```

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)], choix_pondere).
system: f(_21516,_21518)?=f(g(_21522),h(a))|[_21522?=f(_21516)]
decompose: f(_21516,_21518)?=f(g(_21522),h(a))
system: _21516?=g(_21522)|[_21522?=f(_21516),_21518?=h(a)]
expand: _21516?=g(_21522)
system: _21522?=f(g(_21522))|[_21518?=h(a)]
check: _21522?=f(g(_21522))

 No
false.
```

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)], choix_dernier).
system: _23008?=f(_23002)|[f(_23002,_23004)?=f(g(_23008),h(a))]
expand: _23008?=f(_23002)
system: f(_23002,_23004)?=f(g(f(_23002)),h(a))|[]
decompose: f(_23002,_23004)?=f(g(f(_23002)),h(a))
system: _23004?=h(a)|[_23002?=g(f(_23002))]
expand: _23004?=h(a)
system: _23002?=g(f(_23002))|[]
check: _23002?=g(f(_23002))

 No
false.
```

9. Tests

Nous avons effectué des tests unitaires afin de vérifier que les éléments implémentés fonctionne comme on l'attend.

```
?- run_tests.
% PL-Unit: regles ............................. done
% PL-Unit: reduit ..................... done
% PL-Unit: strategy ............. done
% PL-Unit: unifie_ ...... done
% All 71 tests passed
true.
```

Code:

```
include(script).

:- begin_tests(regles).

test(rename) :- regles(X?=T, rename).
test(rename, [fail]) :- regles(a?=b, rename).
test(rename, [fail]) :- regles(X?=a, rename).
test(rename, [fail]) :- regles(f(X)?=a, rename).

test(simplify) :- regles(X?=a, simplify).
test(simplify, [fail]) :- regles(a?=b, simplify).
test(simplify, [fail]) :- regles(X?=T, simplify).
test(simplify, [fail]) :- regles(f(X)?=a, simplify).

test(expand) :- regles(X?=f(Y), expand).
test(expand) :- regles(X?=f(a), expand).
test(expand, [fail]) :- regles(X?=f(X), expand).
test(expand, [fail]) :- regles(f(X)?=X, expand).
test(expand, [fail]) :- regles(a?=f(a), expand).

test(check) :- regles(X?=f(X), check).
test(check, [fail]) :- regles(X?=f(Y), check).
test(check, [fail]) :- regles(f(X)?=X, check).
test(check, [fail]) :- regles(X?=X, check).

test(orient) :- regles(f(X)?=X, orient).
test(orient, [fail]) :- regles(T?=X, orient).
test(orient, [fail]) :- regles(T?=f(X), orient).

test(decompose) :- regles(f(X,Y)?=f(Y,Z), decompose).
```

```
test(decompose, [fail]) :- regles(X?=f(Y,Z), decompose).
test(decompose, [fail]) :- regles(f(X,Y)?=Y, decompose).
test(decompose, [fail]) :- regles(f(X,Y)?=g(Y,Z), decompose).
test(decompose, [fail]) :- regles(f(X,Y)?=f(Y), decompose).

test(clash) :- regles(f(X,Y)?=g(Y,Z), clash).
test(clash) :- regles(f(X,Y)?=f(Y), clash).
test(clash, [fail]) :- regles(f(X,Y)?=f(Y,Z), clash).
test(clash, [fail]) :- regles(X?=f(Y,Z), clash).
test(clash, [fail]) :- regles(f(X,Y)?=Y, clash).

:- end_tests(regles).

:- begin_tests(reduit).

test(vide, [forall(member(R ,[rename, simplify, expand])), true(Q ==
[])]) :- reduit(R, X?=T, [], Q).

test(rename, [true(Q == [T?=a])]) :- reduit(rename, X?=T, [X?=a], Q).
test(rename, [true(Q == [Z?=a])]) :- reduit(rename, X?=T, [Z?=a], Q).
test(rename, [true(Q == [T?=a])]) :- reduit(rename, X?=T, [T?=a], Q).
test(rename, [true(Q == [Z?=T])]) :- reduit(rename, X?=T, [Z?=X], Q).

test(simplify, [true(Q == [a?=T])]) :- reduit(simplify, X?=a, [X?=T],
Q).
test(simplify, [true(Q == [Y?=T])]) :- reduit(simplify, X?=a, [Y?=T],
Q).
test(simplify, [true(Q == [Y?=a])]) :- reduit(simplify, X?=a, [Y?=X],
Q).

test(expand, [true(Q == [f(Y)?=T])]) :- reduit(expand, X?=f(Y), [X?=T],
Q).
test(expand, [true(Q == [Y?=T])]) :- reduit(expand, X?=f(Y), [Y?=T], Q).
test(expand, [true(Q == [Y?=f(Y)])]) :- reduit(expand, X?=f(Y), [Y?=X],
Q).

test(orient, [true(Q == [X?=T])]) :- reduit(orient, T?=X, [], Q).
test(orient, [true(Q == [X?=T, X?=Y])]) :- reduit(orient, T?=X, [X?=Y],
Q).
test(orient, [true(Q == [X?=f(Y)])]) :- reduit(orient, f(Y)?=X, [], Q).
test(orient, [true(Q == [X?=a])]) :- reduit(orient, a?=X, [], Q).

test(decompose, [true(Q == [X?=Y])]) :- reduit(decompose, f(X)?=f(Y),
[], Q).
test(decompose, [true(Q == [X?=Y, X?=T])]) :- reduit(decompose,
f(X)?=f(Y), [X?=T], Q).
```

```
test(decompose, [true(Q == [X?=Y, Y?=Z])]) :- reduit(decompose,
f(X,Y)?=f(Y,Z), [], Q).
test(decompose, [true(Q == [X?=Y, Y?=a])]) :- reduit(decompose,
f(X,Y)?=f(Y,a), [], Q).
test(decompose, [true(Q == [g(X)?=Y, Y?=a])]) :- reduit(decompose,
f(g(X),Y)?=f(Y,a), [], Q).

:- end_tests(reduit).

:- begin_tests(strategy).

test(get_i, [true(I == 1)]) :- get_i(X?=f(Y), I).
test(get_i, [true(I == 2)]) :- get_i(f(X)?=f(Y), I).
test(get_i, [true(I == 3)]) :- get_i(f(T)?=X, I).
test(get_i, [true(I == 4)]) :- get_i(X?=T, I).
test(get_i, [true(I == 4)]) :- get_i(X?=a, I).
test(get_i, [true(I == 5)]) :- get_i(X?=f(X), I).
test(get_i, [true(I == 5)]) :- get_i(f(X)?=g(X), I).

test(choix_eq, [true(RES == [X?=f(X), X?=T, f(X)?=f(Z)])]) :-
choix_eq([f(X)?=f(Z), X?=f(X)], [], X?=T, RES).
test(choix_eq, [true(RES == [f(X)?=g(Z), X?=f(X), X?=T])]) :-
choix_eq([f(X)?=g(Z), X?=f(X)], [], X?=T, RES).
test(choix_eq, [true(RES == [X?=T])]) :- choix_eq([], [], X?=T, RES).

test(last_list, [true(RES == [X?=T])]) :- last_list([X?=T], RES).
test(last_list, [true(RES == [X?=a, X?=T])]) :- last_list([X?=T, X?=a],
RES).
test(last_list, [true(RES == [f(Y) ?= Z, X?=T, X?=a])]) :-
last_list([X?=T, X?=a, f(Y) ?= Z], RES).

:- end_tests(strategy).

:- begin_tests(unifie_).

test(unifie, [forall(member(STRATEGY ,[choix_premier, choix_pondere,
choix_dernier]))]) :- unif([f(X,Y)?= f(g(Z), h(a)), Z ?= f(Y)],
STRATEGY).
test(unifie, [forall(member(STRATEGY ,[choix_premier, choix_pondere,
choix_dernier])), fail]) :- unif([f(X,Y)?= f(g(Z), h(a)), Z ?= f(Y), X
?= f(X)], STRATEGY).

:- end_tests(unifie_).
```

---

10. Code de l'application

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

% Prédicats d'affichage fournis

% set_echo: ce prédicat active l'affichage par le prédicat echo
set_echo :- assert(echo_on).

% clr_echo: ce prédicat inhibe l'affichage par le prédicat echo
clr_echo :- retractall(echo_on).

% 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(_).

%-----------------------------------------------------------------------
%   QUESTION 1.
%-----------------------------------------------------------------------

% Règles

% Rename
% renvoie vrai si T est une variable
regles(X?=T, rename) :- var(X), var(T), !.

% Simplify
% renvoie vrai si T est une constante
regles(X?=T, simplify) :- var(X), atomic(T), !.

% Expand
% renvoi vrai si T est une fonction et X n'apparaît pas dans T
regles(X?=T, expand) :- var(X), compound(T), not(occur_check(X, T)), !.

% Check
% renvoie vrai si X est différent de T et X apparaît dans T
regles(X?=T, check) :- var(X), X\==T , occur_check(X, T), !.

% Orient
% renvoi vrai si T n'est pas une variable
regles(T?=X, orient) :- var(X), nonvar(T), !.
```

```
% Decompose
% renvoi vrai si X et T ont le même symbol et la même arité
regles(F?=G, decompose) :- compound(F), compound(G), functor(F, NAME,
ARITY), functor(G, NAME, ARITY), !.

% Clash
% renvoi vrai si X et T n'ont pas le même symbol ou la meme arité
regles(F?=G, clash) :- compound(F), compound(G), functor(F, FNAME,
FARITY), functor(G, GNAME, GARITY), (FNAME \== GNAME ; FARITY \==
GARITY), !.

% Occur-check
occur_check(V, T) :- var(V), compound(T), contains_var(V, T), !.

%------------------------------------------------------------------------
% Réduits

% Renommage d'une variable
% Renomme toute les occurence de X par la variable T dans P et met le
résultat dans  Q.
reduit(rename, X?=T, P, Q) :- X=T, Q=P.

% Simplification d'une constante
% Renomme toute les occurence de X par la constante T dans P et met le
résultat dans  Q.
reduit(simplify, X?=T, P,Q) :- X=T, Q=P.

% Unification d'une variable avec un terme composé
% Renomme les occurences de X par le terme composé T dans P et met le
résultat dans Q.
reduit(expand, X?=T, P, Q) :- X=T, Q=P.

% Check
% Fail car occur check trouvé par regles(X?=T, check).
reduit(check, _, _, _):- echo("\n No\n"), fail.

% Echange
% Permute le terme T avec la variable X
reduit(orient, T?=X, P, Q) :- append([X?=T], P, Q).

% Décomposition de deux fonctions
% Décompose deux à deux les arguments de deux termes composés.
reduit(decompose, F?=G, P, Q) :- F =.. [_|FARGS], G =.. [_|GARGS],
decomposition(FARGS, GARGS, RES), append(RES, P, Q).

% Clash
```

```
% Fail car clash trouvé par regles(F?=G, clash).
reduit(clash, _, _, _):- echo("\n No\n"), fail.

% Ajoute la relation ?= entre chaque éléments des deux listes données.
decomposition([H1|T1], [H2|T2], RES) :- decomposition(T1, T2, ACC),
append([H1?=H2], ACC, RES).
decomposition([], [], []).

%-------------------------------------------------------------------------

unifie_([H|T], S) :- aff_sys(H|T), regles(H, R), aff_regle(H, R),
reduit(R, H, T, Q), unifie(Q, S).
unifie_(fin) :- echo("\n Yes\n").

%-------------------------------------------------------------------------
%   QUESTION 2.
%-------------------------------------------------------------------------

% Associe un poids à une équation en fonction de la première règle
applicable à cette dernière.
get_i(E, I) :- (regles(E, clash); regles(E, check)) -> I = 5, !;
            (regles(E, rename); regles(E, simplify)) -> I = 4, !;
            regles(E, orient) -> I = 3, !;
            regles(E, decompose) -> I = 2, !;
            regles(E, expand) -> I = 1, !.

% Compare deux à deux le poids associé aux équations afin d'obtenir
celle de poids le plus élévé.
choix_eq([H|T], TMP, E, RES) :- get_i(E, I1), get_i(H, I2),
                                (I1>=I2 -> append([H], TMP,
NEWTMP),choix_eq(T, NEWTMP, E, RES);
                                      (choix_eq(T, [E|TMP], H, RES))
                                ).
choix_eq([], TMP, E, RES) :- append([E], TMP , RES).

% Prend le dernier élément de la liste et le met au début.
last_list(P, [LAST|RESTE]) :- reverse(P, [LAST|R]), reverse(R, RESTE).

%-------------------------------------------------------------------------
% Unification avec les différentes stratégies.


unifie(P, choix_premier) :- unifie_(P, choix_premier), !.

unifie([H|T], choix_pondere) :- choix_eq(T, [], H, Q), unifie_(Q,
choix_pondere), !.
```

```
unifie(P, choix_dernier) :- last_list(P, RES), unifie_(RES,
choix_dernier), !.

unifie([], _) :- unifie_(fin).

%-----------------------------------------------------------------------
%   QUESTION 3.
%-----------------------------------------------------------------------

% Désactive l'affichage de la trace.
unif(P,S) :- clr_echo, unifie(P,S).

% Active l'affichage de la trace.
trace_unif(P,S) :- set_echo, unifie(P,S).

%-----------------------------------------------------------------------
% Prédicat pour l'affichage
aff_sys(P) :- echo('system: '),echo(P),echo('\n').
aff_regle(E,R) :- echo(R),echo(': '),echo(E),echo('\n').


:- initialization(main).

main :- set_echo,
      write("\n\nAlgorithme d\'unification de Martelli-Montanari\n\n"),
      write("P est un système d'équation du type [X1?=Y1, ... ,Xn?=Yn]
où Xi, Yi peuvent être des termes (f(X)), des variables (X) ou des
constantes (a).\n\n"),
      write("S est la stratégie d'unification
souhaitée:\nchoix_premier\nchoix_pondere\nchoix_dernier\n\n"),
      write("Par défaut l'affichage des étapes est activé.\n\n")  ,
      write("Pour désactiver l'affichage des étapes et unifier,
saisir:\ntrace_unif(P,S).\n\n"),
      write("Pour activer l'affichage des étapes et unifier,
saisir:\nunif(P,S).\n\n"),
      write("Pour simplement unifier, saisir:\nunifie(P,S).\n\n\n").
```

11. Sources

Documentation SWI-Prolog
https://www.swi-prolog.org/

Aide technique
https://stackoverflow.com/