Projet LMC - Programmation Logique

Jofrey Luc Quentin Sonrel

12 décembre 2016

Question 1

Implémentation des règles

Utilisation de X ?=T au lieu de E

Pour l'implémentation des règles, dans leur "prototype", nous avons fait le choix d'utiliser `X?=T` pour désigner l'expression en paramètre du prédicat au lieu d'une simple variable `E`, comme suit (exemple) :

```
regle(X?=T, rename)
```

Au lieu de :

```
regle(E, rename)
```

Cela nous permet de simplifier l'implémentation des prédicats en évitant l'utilisation du prédicat `arg` de Prolog.
Ainsi, la règle *rename* peut se définir comme ceci :

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

Au lieu de :

```
regle(E, rename) :- arg(1, E, X), arg(2, E, T), var(X),var(T), !.
```

Cela nous permet aussi d'éviter les erreurs de l'interpréteur (erreur de syntaxe) avec certaines expressions.

Il existe aussi une solution alternative qui consiste à utiliser `functor(E, ?=, 2)}` en début de prédicat, à la place de `arg`.

Implémentation de la règle *clash*

Pour implémenter la règle *clash* nous avons essayé une première approche qui consistait à comparer l'arité et les noms des termes à l'aide du prédicat functor afin déclencher le *clash* dans les cas suivants :

— Arité différente mais noms identiques
— Noms différents mais arité identique

— Arité et noms différents

Le premier jet d'implémentation de cette méthode était pensé pour fonctionner en 3 prédicats (un pour chacun des trois cas cités précédemment).

Cette solution n'a pas été retenue car une alternative plus simple et plus "maligne" était de faire appel à la règle *decompose*, comme ceci :

```
regle(E, clash) :- \+ regle(E, decompose), !.
```

Ce choix est dû au fait que *decompose* compare les noms et l'arité des termes de l'expression, ainsi, si la règle *decompose* n'est pas applicable, *clash* est déclenchable.

Implémentation de occur_check

Notre première tentative d'implémentation du prédicat `occur_check` nous a posé un problème majeur. L'idée de base derrière notre implémentation était, pour une expression, de la parcourir et d'éclater les termes composés (ex : $f(g(X))$) au fur et à mesure afin "d'aplatir" l'expression jusqu'à isoler le terme dont on veut vérifier la présence.

Le procédé d'éclatement était récursif : on prend une expression, on l'éclate à l'aide du prédicat `=..`, on retire le terme non composé issu de l'éclatement et on recommence sur le reste. Par exemple, pour $f(g(X))$, la première étape sépare f et $g(X)$ afin de relancer le processus sur $g(X)$ après avoir "mis de côté" (liste séparée) f. Le but est donc d'aplatir une expression telle que $f(g(X))$ en une liste $[f, g, X]$ (par exemple), afin de faciliter la recherche d'un terme.

Le problème rencontré dans l'implémentation de cette méthode est le cas d'une tentative d'éclatement d'un terme non composé. En effet, lors du parcours, si on rencontre un terme non composé il sera impossible d'éclater avec `=..`, causant une erreur et donc l'arrêt de la récursion, le prédicat ne fonctionne donc pas...

Au cours de nos recherches pour résoudre ce soucis, nous avons découvert l'existence du prédicat `contains_var` qui permet justement de vérifier si une expression contient un terme donné. Nous avons donc décidé d'utiliser cette alternative qui correspondait parfaitement à nos besoin.

Implémentation de la réduction

Lors de l'implémentation de la réduction, le problème majeur rencontré était celui du remplacement de variable (via le prédicat `remplace`). Lors de notre première tentative d'implémenter le prédicat `remplace`, celui-ci ne fonctionnait que sur un remplacement d'une variable par un atome mais pas sur le remplacement d'une variable par une autre (notre objectif ici). La procédure était basée sur un exemple trouvé sur internet [1].

Après quelques essais et recherches, nous avons réussi à obtenir un prédicat fonctionnel pour le remplacement d'une variable par une autre, notamment à l'aide de l'utilisation

1. http://stackoverflow.com/questions/12638347/replace-atom-with-variable/37526158#37526158

du prédicat == pour la substitution de variable. Cette seconde méthode fonctionnait avec des appels récursifs à `remplace` avec association d'une expression et de sa version éclatée en liste (méthode similaire à celle employée pour la première implémentation de notre `occur_check`). Le but état d'aplatir l'expression en liste jusqu'à trouver la variable à remplacer, la remplacer, puis rassembler le tout dans le format d'origine.

Cette méthode ne fonctionnait hélas que sur des expression simples (ex : $f(g(X))$) mais pas sur des expressions plus complexes (ex : $f(g(X)) + h(X)$).

La solution à ce problème a été une grosse simplification du problème avec l'utilisation directe du prédicat = pour le remplacement de variable. Cette méthode n'est pas idéale dans la mesure où ce prédicat fait appel à une unification (native à Prolog) alors que notre but est ici d'implémenter une unification...

Question 2

Cette question portait sur l'implémentation de différentes stratégies pour l'exécution de l'algorithme d'unification. Le travail réalisé s'est donc découpé en deux parties : l'implémentation d'une stratégie alternative à celle employée jusqu'à présent, et l'implémentation de prédicats permettant de choisir quelle stratégie employer.

Ces implémentations ont donc donné naissance à plusieurs prédicats dont les prédicats `unifie(P, choix_premier)` et `unifie(P, choix_pondere)` utilisant respectivement la stratégie "simple" (traitement des équations dans l'ordre) et la stratégie "pondérée" (choix d'une équation à traiter en fonction du poids de la règle à lui appliquer). Chacune de ces deux stratégies ont demandé l'écriture et l'utilisation des prédicats `choix_premier(P, Q, E, R)` et `choix_pondere(P, Q, E, R)` qui servent à choisir les équations à traiter dans le bon ordre pour chaque stratégie.

Implémentation des différentes stratégies

choix_premier

L'implémentation de `choix_premier` est la plus simple car elle fait fonctionner l'algorithme d'unification de la même façon que précédemment (en utilisant l'ordre naturel des équations). Ainsi, pour chacune des règles, `choix_premier` appelle simplement les prédicats d'unification écrits dans la question 1.

Le prédicat `unifie(P, choix_premier)` appelle directement `unifie(P)` comme précédemment puisque c'est un cas particulier de `unifie(P, S)` où S = choix_premier.

choix_pondere

L'implémentation de `choix_pondere` consiste en 8 prédicats, un par règle (plus un prédicat de fin lorsuqe plus aucune règle n'est applicable), ordonnés dans le programme par poids (*clash/check* > *rename/simplify* > *orient* > *decompose* > *expand*) et se terminant chacun par le `cut`. Chacun de ces prédicats utilise deux sous-prédicats :

`premiere_applicable(P, R, E)`, qui renvoie dans E la première équation de P qui satisfait R, puis `retirer(E, P, Q)`, qui renvoie dans Q l'ensemble P sans l'équation R. Au final, un appel à `choix_pondere(P, Q, E, R)` renverra dans R la règle avec la priorité la plus haute qui peut être appliquée sur au moins une des équations de P, dans E la première équation sur laquelle peut être appliquée R, et dans Q P\E.

De ce fait, `unifie(P, choix_pondere)` va quant à lui appliquer successivement sur P `choix_pondere`, puis sur les E et Q obtenus `unifie_pondere(R, E, Q, Suite)`, une version alternative de `unifie(P)` qui va simplement faire les affichages puis appliquer `reduit` sur R, E, Q et Suite (contrairement à `unifie` qui l'appelait directement sur la première équation de P).

`unifie(P, choix_pondere)` va ensuite s'appeler récursivement sur la liste d'équations réduite Suite jusqu'à s'arrêter sur la liste vide.

Comparatif des stratégies

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)], choix_premier).
system: [f(_G201,_G202)?=f(g(_G204),h(a)),_G204?=f(_G201)]
decompose: f(_G201,_G202)?=f(g(_G204),h(a))
system: [_G3?=f(_G1),_G2?=h(a),_G1?=g(_G3)]
expand: _G3?=f(_G1)
system: [_G2?=h(a),_G1?=g(f(_G1))]
expand: _G2?=h(a)
system: [_G1?=g(f(_G1))]
check: _G1?=g(f(_G1))
```

FIGURE 1 – Trace d'exécution avec choix_premier

```
?- trace_unif([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)], choix_pondere).
system: [f(_G242,_G243)?=f(g(_G245),h(a)),[_G245?=f(_G242)]]
decompose: f(_G242,_G243)?=f(g(_G245),h(a))
system: [_G245?=f(_G242),[_G243?=h(a),_G242?=g(_G245)]]
expand: _G245?=f(_G242)
system: [_G242?=g(f(_G242)),[_G243?=h(a)]]
check: _G242?=g(f(_G242))
```

FIGURE 2 – Trace d'exécution sur choix_pondere

On constate ici une légère différence entre l'exécution des deux stratégies : `choix_pondere` effectue une étape de moins. On peut donc en conclure que `choix_pondere` est plus rapide à l'exécution que `choix_premier`. Cela est dû au fait que `choix_pondere` ordonne les étapes en mettant *clash* et *check* en premier. Ces deux étapes étant critiques et servant de cas d'arrêt à l'algorithme, cela permet de s'arrêter plutôt et de sauter des étapes inutiles, rendant l'algorithme plus rapide.
Il est à noter que dans certains cas l'exécution sera la même entre les deux stratégies.

Cela rend donc `choix_pondere` plus rapide en moyenne seulement (au pire aussi lent, au mieux plus rapide que `choix_premier`).

Autre stratégie possible

Une autre alternative à choix_premier et à choix_pondere serait de donner la priorité aux règles *clash* et *check* (donc les appliquer dès que possible) et sinon choisir la première équation de la liste.

Question 3

Dans cette question, nous avons implémenté deux prédicats simples, le premier permet de lancer l'algorithme d'unification précédemment écrit en affichant une trace d'exécution :

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

Tandis que le second, permet de lancer le même algorithme mais cette fois sans afficher la trace d'exécution :

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

Exemples et traces d'exécution

Exemple sur une autre équation

```
?- trace_unif([X?=Y, Y?=g(h(Z)), X?=Z], choix_premier).
system: [_G201?=_G202,_G202?=g(h(_G207)),_G201?=_G207]
rename: _G201?=_G202
system: [_G201?=g(h(_G207)),_G201?=_G207]
expand: _G201?=g(h(_G207))
system: [g(h(_G207))?=_G207]
orient: g(h(_G207))?=_G207
system: [_G207?=g(h(_G207))]
check: _G207?=g(h(_G207))
false.
```

FIGURE 3 – Nouvel exemple en stratégie choix_premier

```
?- trace_unif([X?=Y, Y?=g(h(Z)), X?=Z], choix_pondere).
system: [_G201?=_G202,[_G202?=g(h(_G207)),_G201?=_G207]]
rename: _G201?=_G202
system: [_G201?=_G207,[_G201?=g(h(_G207))]]
rename: _G201?=_G207
system: [_G201?=g(h(_G201)),[]]
check: _G201?=g(h(_G201))
false.
```

FIGURE 4 – Même exemple en stratégie choix_pondere

Notes :
— On constate que l'unification échoue, comme il se doit.
— La trace d'exécution est cohérente.
— La stratégie choix_pondere est à nouveau plus rapide/efficace que choix_premier.

Code source

```
1  :- op(20,xfy,?=).
2  set_echo :- assert(echo_on).
3  clr_echo :- retractall(echo_on).
4  echo(T) :- echo_on, !, write(T).
5  echo(_).
6
7  %%% Question 1 %%%
8
9  %% Régles %%
10
11 regle(X?=T, rename) :-
12     % Rename applicable si X variable et T variable
13     var(X),
14     var(T), !.
15
16 regle(X?=T, simplify) :-
17     % Simplify applicable si X variable et T constante
18     var(X),
19     atomic(T), !.
20
21 regle(X?=T, expand) :-
22     % Expand applicable si X variable et T composé (et pas une liste)..
23     var(X),
24     compound(T),
25     \+ is_list(T),
26     % .. et si T ne contient pas X
27     \+ occur_check(X, T), !.
28
29 regle(X?=T, check) :-
30     % Check applicable si X variable...
31     var(X),
32     % ...et X différent de X...
33     X \== T,
34     % ...et si T contient X
35     occur_check(X, T), !.
36
37
38 regle(T?=X, orient) :-
39     % Orient applicable si X variable et T non variable
40     var(X),
41     nonvar(T), !.
42
43 regle(Fgauche?=Fdroite, decompose) :-
44     % Decompose applicable si les deux termes sont composés (et pas des
           listes)..
45     compound(Fgauche),
46     compound(Fdroite),
47     \+ is_list(Fgauche),
48     \+ is_list(Fdroite),
49     % .. et si ils ont le même nom et la même arité
50     functor(Fgauche, Nom, Arite),
51     functor(Fdroite, Nom, Arite), !.
52
53 regle(Fgauche?=Fdroite, clash) :-
```

```
54      % Clash applicable si les deux termes sont composés (et pas des listes)
            ..
55      compound(Fgauche),
56      compound(Fdroite),
57      \+ is_list(Fgauche),
58      \+ is_list(Fdroite),
59      % .. et dans les conditions contraires à decompose
60      \+ regle(Fgauche?=Fdroite, decompose), !.
61
62  %% Occur_check %%
63
64  occur_check(X, T) :- contains_var(X, T).
65
66  %% Reduit %%
67
68  reduit(rename, X?=T, P, Q) :-
69      % On unifie X avec T, et Q vaut la liste d'équations ainsi modifiée
70      X = T,
71      Q = P.
72
73  reduit(simplify, X?=T, P, Q) :-
74      % On unifie X avec T, et Q vaut la liste d'équations ainsi modifiée
75      X = T,
76      Q = P.
77
78  reduit(expand, X?=T, P, Q) :-
79      % On unifie X avec T, et Q vaut la liste d'équations ainsi modifiée
80      X = T,
81      Q = P.
82
83  reduit(orient, X?=T, P, Q) :-
84      % On ajoute l'équation inversée à la liste de base, et on stocke le ré
            sultat dans Q
85      append(P, [T?=X], Q).
86
87  reduit(decompose, Fgauche?=Fdroite, P, Q) :-
88      % On prend les arguments des fonctions...
89      Fgauche =.. [_|ListeFgauche],
90      Fdroite =.. [_|ListeFdroite],
91      % ...et on les fusionne deux à deux pour en faire des équations a?=b
92      decompose_fusion(ListeFgauche, ListeFdroite, Resultat),
93      % On ajoute la liste des a?=b à la tête de Q (P'), la queue de Q (S) ne
            change pas.
94      append(P, Resultat, Q).
95
96  decompose_fusion([],[],[]) :- !. % Cas d'arrêt, lorsque les deux listes
        sont vides l'équation est nulle
97
98  decompose_fusion([TeteGauche|QueueGauche], [TeteDroite|QueueDroite], R) :-
99      % Récursion sur la queue des listes : on construit petit à petit la
            liste d'équations
100     decompose_fusion(QueueGauche, QueueDroite, Rsuivant),
101     append(Rsuivant, [TeteGauche?=TeteDroite], R).
102
103 %% Unifie %%
104
105 % Cas d'arrêt, liste vide
106 unifie([]) :- !.
```

```
107
108 % Applique rename si possible
109 unifie([HeadP|TailP]) :-
110     regle(HeadP, rename), !,
111     echo('system: '),
112     echo([HeadP|TailP]), echo('\n'),
113     echo('rename: '),
114     echo(HeadP), echo('\n'),
115     reduit(rename, HeadP, TailP, Q),
116     unifie(Q), !.
117
118 % Applique simplify si possible
119 unifie([HeadP|TailP]) :-
120     regle(HeadP, simplify), !,
121     echo('system: '),
122     echo([HeadP|TailP]), echo('\n'),
123     echo('simplify: '),
124     echo(HeadP), echo('\n'),
125     reduit(simplify, HeadP, TailP, Q),
126     unifie(Q), !.
127
128 % Applique expand si possible
129 unifie([HeadP|TailP]) :-
130     regle(HeadP, expand), !,
131     echo('system: '),
132     echo([HeadP|TailP]), echo('\n'),
133     echo('expand: '),
134     echo(HeadP), echo('\n'),
135     reduit(expand, HeadP, TailP, Q),
136     unifie(Q), !.
137
138 % Applique check si possible
139 unifie([HeadP|TailP]) :-
140     regle(HeadP, check), !,
141     echo('system: '),
142     echo([HeadP|TailP]), echo('\n'),
143     echo('check: '),
144     echo(HeadP), echo('\n'),
145     fail, !.
146
147 % Applique orient si possible
148 unifie([HeadP|TailP]) :-
149     regle(HeadP, orient), !,
150     echo('system: '),
151     echo([HeadP|TailP]), echo('\n'),
152     echo('orient: '),
153     echo(HeadP), echo('\n'),
154     reduit(orient, HeadP, TailP, Q),
155     unifie(Q), !.
156
157 % Applique decompose si possible
158 unifie([HeadP|TailP]) :-
159     regle(HeadP, decompose), !,
160     echo('system: '),
161     echo([HeadP|TailP]), echo('\n'),
162     echo('decompose: '),
163     echo(HeadP), echo('\n'),
164     reduit(decompose, HeadP, TailP, Q),
```

```
165     unifie(Q), !.
166
167 % Applique clash si possible
168 unifie([HeadP|TailP]) :-
169     regle(HeadP, clash), !,
170     echo('system: '),
171     echo([HeadP|TailP]), echo('\n'),
172     echo('clash: '),
173     echo(HeadP), echo('\n'),
174     fail, !.
175
176 % Lorsque plus aucune régle n'est applicable, finit
177 unifie(_) :- !.
178
179 %%% Question 2 %%%
180
181 % Choix_premier (pas utilisé) : indique simplement si on peut appliquer une
        régle sur la premiére équation de P
182 choix_premier([HeadP|TailP], TailP, HeadP, rename) :- regle(HeadP, rename),
        !.
183 choix_premier([HeadP|TailP], TailP, HeadP, simplify) :- regle(HeadP,
    simplify), !.
184 choix_premier([HeadP|TailP], TailP, HeadP, expand) :- regle(HeadP, expand),
        !.
185 choix_premier([HeadP|TailP], TailP, HeadP, check) :- regle(HeadP, check),
    !.
186 choix_premier([HeadP|TailP], TailP, HeadP, orient) :- regle(HeadP, orient),
        !.
187 choix_premier([HeadP|TailP], TailP, HeadP, decompose) :- regle(HeadP,
    decompose), !.
188 choix_premier([HeadP|TailP], TailP, HeadP, clash) :- regle(HeadP, clash),
    !.
189
190 % unifie(P) cas particulier de unifie(P, S)
191 unifie(P, choix_premier) :- unifie(P), !.
192
193 % choix_pondere %
194
195 % Renvoie la premiére équation E de P qui satisfait la régle R
196 premiere_applicable([HeadP|_], R, E) :-
197     regle(HeadP, R), % Si la tête de P satisfait R, on l'unifie avec E
198     E = HeadP, !.
199 premiere_applicable([_|TailP], R, E) :-
200     premiere_applicable(TailP, R, E). % Sinon, on regarde la queue de P
201
202 % Enléve l'équation E de P et met le reste P\{E} dans Q
203 retirer(E, [HeadP|TailP], TailP):-
204     E == HeadP, !. % Si E est la tête de P, on renvoie la queue de P
205 retirer(E, [HeadP|TailP], Q):-
206     retirer(E, TailP, Q1),
207     append([HeadP], Q1, Q). % Sinon, on concaténe la tête de P avec le ré
        sultat de l'appel récursif
208
209 % choix_pondere : grâce au cut, on évaluera successivement tous les pré
    dicats jusqu'à en trouver un applicable, puis on retournera dans unifie
210 choix_pondere(P, Q, E, clash) :-
211     premiere_applicable(P, clash, E),
212     retirer(E, P, Q), !.
```

```
213 choix_pondere(P, Q, E, check) :-
214     premiere_applicable(P, check, E),
215     retirer(E, P, Q), !.
216 choix_pondere(P, Q, E, rename) :-
217     premiere_applicable(P, rename, E),
218     retirer(E, P, Q), !.
219 choix_pondere(P, Q, E, simplify) :-
220     premiere_applicable(P, simplify, E),
221     retirer(E, P, Q), !.
222 choix_pondere(P, Q, E, orient) :-
223     premiere_applicable(P, orient, E),
224     retirer(E, P, Q), !.
225 choix_pondere(P, Q, E, decompose) :-
226     premiere_applicable(P, decompose, E),
227     retirer(E, P, Q), !.
228 choix_pondere(P, Q, E, expand) :-
229     premiere_applicable(P, expand, E),
230     retirer(E, P, Q), !.
231 choix_pondere(_, [], [], []) :- !. % Si aucune régle applicable : on
    renvoie des listes vides (cas d'arrêt)
232
233 % unifie(P, choix_pondere)
234
235 unifie([], choix_pondere) :- !. % Cas d'arrêt sur liste vide
236
237 % Récursion
238 unifie(P, choix_pondere) :-
239     choix_pondere(P, Q, E, R), % On cherche quelle régle appliquer sur
        quelle équation
240     unifie_pondere(R, E, Q, Suite), % On applique cette régle sur cette é
        quation
241     unifie(Suite, choix_pondere). % Appel récursif sur la liste d'équations
         modifiée
242
243 % Applique rename
244 unifie_pondere(rename, E, Q, Suite) :-
245     echo('system: '),
246     echo([E,Q]), nl,
247     echo('rename: '),
248     echo(E), nl,
249     reduit(rename, E, Q, Suite).
250
251 % Applique simplify
252 unifie_pondere(simplify, E, Q, Suite) :-
253     echo('system: '),
254     echo([E,Q]), nl,
255     echo('simplify: '),
256     echo(E), nl,
257     reduit(simplify, E, Q, Suite).
258
259 % Applique expand
260 unifie_pondere(expand, E, Q, Suite) :-
261     echo('system: '),
262     echo([E,Q]), nl,
263     echo('expand: '),
264     echo(E), nl,
265     reduit(expand, E, Q, Suite).
266
```

```
267 % Applique check
268 unifie_pondere(check, E, Q, Suite) :-
269     echo('system: '),
270     echo([E,Q]), nl,
271     echo('check: '),
272     echo(E), nl,
273     reduit(check, E, Q, Suite).
274
275 % Applique orient
276 unifie_pondere(orient, E, Q, Suite) :-
277     echo('system: '),
278     echo([E,Q]), nl,
279     echo('orient: '),
280     echo(E), nl,
281     reduit(orient, E, Q, Suite).
282
283 % Applique decompose
284 unifie_pondere(decompose, E, Q, Suite) :-
285     echo('system: '),
286     echo([E,Q]), nl,
287     echo('decompose: '),
288     echo(E), nl,
289     reduit(decompose, E, Q, Suite).
290
291 % Applique clash
292 unifie_pondere(clash, E, Q, Suite) :-
293     echo('system: '),
294     echo([E,Q]), nl,
295     echo('clash: '),
296     echo(E), nl,
297     reduit(clash, E, Q, Suite).
298
299 % Cas d'arrêt (aucune régle applicable)
300 unifie_pondere([], [], [], []).
301
302 %%% Question 3 %%%
303
304 unif(P, S) :- clr_echo, unifie(P, S). % Affichage sans trace
305 trace_unif(P, S) :- set_echo, unifie(P, S). % Affichage avec trace
```

Sources

— Manuel SWI Prolog (prise en main du langage, notions globales)
— Documentation SWI Prolog (documentation des prédicats)
— StackOverflow (exemples, cas similaires, etc...)