TP - LMC - Master Informatique M1 2010/2011

1 Description du sujet

Le but du TP est d'implanter en PROLOG une variante de l'algorithme d'unification de Martelli-Montanari vu en cours et dont les règles de transformation sont les suivantes :

Rename $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow P'[x/t]; S[x/t] \cup \{x = t\}$ si t est une variable

Simplify $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow P'[x/t]; S[x/t] \cup \{x = t\}$ si t est une constante

Expand $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow P'[x/t]; S[x/t] \cup \{x = t\}$ si t est composé et x n'apparaît pas dans t

Check $\{x \stackrel{?}{=} t\} \cup P'; S \rightsquigarrow \perp$ si $x \neq t$ et x apparaît dans t

Orient $\{t \stackrel{?}{=} x\} \cup P'; S \rightsquigarrow \{x \stackrel{?}{=} t\} \cup P'; S$ si t n'est pas une variable

Decompose $\{f(s_1, \cdots, s_n) \stackrel{?}{=} f(t_1, \cdots, t_n)\} \cup P'; S \rightsquigarrow \{s_1 \stackrel{?}{=} t_1, \cdots, s_n \stackrel{?}{=} t_n\} \cup P'; S$

Clash $\{f(s_1, \cdots, s_n) \stackrel{?}{=} g(t_1, \cdots, t_m)\} \cup P'; S \rightsquigarrow \perp$ si $f \neq g$ ou $m \neq n$

On rappelle que l'algorithme de Martelli-Montanari opère sur des paires $P; S$ où P est un ensemble (système) d'équations $s \stackrel{?}{=} t$ à unifier et S est un ensemble d'équations $s = t$ déjà résolues. Le symbole $\perp$ désigne le système qui n'a pas de solution. On note x pour désigner une variable et t pour désigner un terme quelconque (variable, constante ou terme composé).
La notation $P[x/t]$ exprime la substitution simultanée de toutes les occurences de la variable x par le terme t dans un ensemble P.
On rappelle également que l'algorithme démarre initialement avec une paire $P; \emptyset$ où P est le système à résoudre et applique ensuite les règles autant que possible. Lorsque plus aucune règle ne peut s'appliquer, on obtient dans S l'unificateur le plus général (mgu) du système P ou $\perp$ si le système n'admet pas d'unificateur.

1.1 Question 1

Dans un premier temps, il s'agit d'écrire un prédicat `unifie(P)` où `P` est un système d'équations à résoudre représenté sous la forme d'une liste `[S1 ?= T1,...,SN ?= TN]`. La définition de l'opérateur `?=` vous est fournie (`:- op(20,xfy,?=)`).
Voici deux exemples d'utilisation du prédicat `unifie` :

```
?- unifie([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(Y)]).
system: [f(_G496, _G497) = f(g(_G499), h(a)), _G499 = f(_G497)]
decompose: f(_G496, _G497) = f(g(_G499), h(a))
system: [_G496 = g(_G499), _G497 = h(a), _G499 = f(_G497)]
expand: _G496 = g(_G499)
system: [_G497 = h(a), _G499 = f(_G497)]
expand: _G497 = h(a)
system: [_G499 = f(h(a))]
expand: _G499 = f(h(a))

X = g(f(h(a)))
Y = h(a)
Z = f(h(a))

Yes
```

```
?- unifie([f(X,Y) ?= f(g(Z),h(a)), Z ?= f(X)]).
system: [f(_G496, _G497) = f(g(_G499), h(a)), _G499 = f(_G496)]
decompose: f(_G496, _G497) = f(g(_G499), h(a))
system: [_G496 = g(_G499), _G497 = h(a), _G499 = f(_G496)]
expand: _G496 = g(_G499)
system: [_G497 = h(a), _G499 = f(g(_G499))]
expand: _G497 = h(a)
system: [_G499 = f(g(_G499))]
check: _G499 = f(g(_G499))

No
```

Pour réaliser `unifie(P)`, on implantera les prédicats suivants :

- `regle(E,R)` : détermine la règle de transformation `R` qui s'applique à l'équation `E`, par exemple, le but `?- regle(f(a) ?= f(b),decompose)` réussit.
- `occur_check(V,T)` : teste si la variable `V` apparaît dans le terme `T`.
- `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`.

1.2 Question 2

L'efficacité de l'algorithme d'unification dépend beaucoup de l'ordre dans lequel on choisit les équations à résoudre. Une méthode particulière de choix de l'équation à résoudre à chaque étape s'appelle une *stratégie*. Une stratégie peut donc s'implanter comme un prédicat `choix(P,Q,E,R)` qui choisit dans le système `P` une équation `E` avec sa règle correspondante `R` et l'extrait de `P` pour donner le système `Q`.
La stratégie utilisée dans les exemples précédents est celle qui choisit systématiquement la première équation de la liste, appelons cette stratégie `choix_premier`. Une autre stratégie possible est celle qui affecte un poids à chaque règle et privilégie celles qui ont le poids le plus élevé, appelons la `choix_pondere`.
Une échelle de poids possible est la suivante :

$$clash, check > rename, simplify > orient > decompose > expand$$

On souhaite à présent permettre l'implantation et le paramétrage de plusieurs stratégies. Pour celà, vous devez écrire un prédicat `unifie(P,S)` où `S` est le nom de la stratégie à mettre en œuvre. Ainsi, le prédicat `unifie(P)` devient un cas particulier de `unifie(P,S)` où `S = choix_premier`. On demande d'implanter les prédicats `choix_premier(P,Q,E,R)` et `choix_pondere(P,Q,E,R)` et de proposer d'autres stratégies possibles. Un point important consistera à comparer sur des exemples appropriés ces différentes stratégies.

1.3 Question 3

A l'aide des prédicats `set_echo` et `clr_echo` qui vous sont fournis, implanter deux prédicats `unif(P,S)` et `trace_unif(P,S)` qui permettent respectivement d'inhiber ou d'activer la trace d'affichage des règles appliquées à chaque étape.

2 Travail demandé (à rendre pour le 17/01/2011)

- un petit rapport (quelques pages) détaillant les réponses aux questions avec une description claire et une justification de vos choix d'implantation,
- un listing *commenté* des sources PROLOG de votre programme,
- des traces d'exécution de votre programme sur des exemples significatifs.

Il faudra bien-sûr que le système implanté soit tel que son utilisation soit facile et pratique, ce qui sera testé lors de la soutenance, et donc les aspects liés à l'interface utilisateur ne doivent pas être négligés.

Pour réaliser le TP, vous pourrez utiliser les prédicats prédéfinis suivants (liste non exhaustive) : `functor`, `arg`, `=..`, `atomic`, `compound`, `var`, `nonvar`, `call`... Ces prédicats sont décrits dans la documentation en ligne de PROLOG, par exemple, la commande `?- help(functor).` permet d'obtenir la description du prédicat `functor`.