CHAPITRE III

Extensions de la Programmation en Logique

Ecole Centrale de Lyon
Département Mathématiques,
Informatiques et Systèmes
1993-94
S. SAIDI

Extensions de la Programmation en Logique

- **Extensions Constraintes**
  - CLP

- **Extensions Grammaticales**
  - Grammaires Logiques
    - DCG
    - AFFLOG

- **Extensions Fonctionnelles, Objets**
  - LIFE

- **Extensions Objets**
  - L & O

- **Extensions BD**
  - BD Relationnelles, Couplage, ...

Extensions Objet, Héritage, ...

LOGIN : Logic with Inheritance

- **Héritage** : moyen de capturer une information relevant d'une classe par une relation "EST-UN"

Exemple1 : "Les tigres sont des mammifères"

➥ toutes les propriétés des mammifères sont héritées par les tigres.

- Augmente la capacité expressive du langage.

- Les travaux dans ce domaine ont tendance à alourdir l'aspect syntaxique (de Prolog).

  L'approche *réseaux sémantiques* n'a pas une sémantique claire.

- Deux manières de mettre en place l'héritage :
  1 - Par unification (intégration dans l'unification)
  2 - Par la résolution, codage en prédicats
  souvent employée par les méta-interprètes

- "Le tigre EST-UN mammifère"
  ==> "$\forall x$, tigre(x) $\Rightarrow$ mammifère(x)"

  L'héritage des mammifères $\rightarrow$ tigres constitue une étape d'inférence, et non de déduction.
  ➥ Appliquer la résolution rend la preuve plus longue.

  L'idée est d'écourter la résolution (réduire la preuve) et d'intégrer cette inférence dans l'unification.

  Question des retour-arrières (héritage multiple)

- Sachant que l'unification classique de Robinson est un cas particulier de l'unification des termes d'une algèbre 'multi-sorte';
  LOGIN propose une approche basée sur les treillis de types et procure une extension simple de l'unification
  ➥ Séparation de l'héritage (multiple) de la machine Prolog.

- L'approche proposée considère le type comme une abstraction d'ensemble .
  ➥ Notions de GLB, LUB

Exemple2 :

" Toute personne s'aime ;
les élèves sont des personnes ;
jean est un étudiant "

➠ Codage en logique de 1er ordre :

$\forall x$, personne (x) $\Rightarrow$ aime (x, x)
& $\forall x$, élève (x) $\Rightarrow$ personne (x)
& élève (jean)

➠ En Prolog :

```
aime (x, x) : - personne (x)
personne (x) : - élève (x)
élève (jean).
```

Pour vérifier si Jean s'aime :

? - aime (jean, jean). $\Rightarrow$ oui

➠ On peut également avoir la représentation typée :

$\forall x \in$ personne, aime (x, x)
& élève $\subset$ personne
& jean $\in$ élève

➥ On peut vérifier plus efficacement les types (i.e. un ensemble est sous-ensemble d'un autre, un élément appartient à un ensemble).

➥ On peut vérifier que *Jean s'aime* par l'application d'un *Modus-Ponens* plutôt que deux (ou *Modus-Tolens* c.f. Prolog).

Exemple3 :

Un étudiant est une personne ;
$p_1, \ldots, p_n$ sont des personnes ;
$s_1, \ldots, s_n$ sont des étudiants ;
p_i et s_j ont une certaine propriété Prop, $1 \leq i \leq n$, $1 \leq j \leq m$.

➠ En prolog :

```
personne(X) :- etudiant(X).
personne (p1).
        ...
personne (pn).
étudiant (si).
        ...
étudiant (sm).
prop (pi).
prop (sj).
```

Question :

?- personne (X), prop (X).

➥ X = p_i, X= s_j

Beaucoup de modus-tolens

- Il est simple de voir que s_j est une personne puisqu'elle est un étudiant.

  On "déclare" que dans l'unification, le symbole *étudiant* "s'unifie" avec le symbole *personne* :
  étudiant <| personne où '<| ' : EST-UN

  personne := {p_1; ... ; p_n}.
  étudiant := {s_1 ; ... ; s_m}.

et les faits

prop (p_i).
prop (p_j).

et on écrit

prop (x : personne) ? /* x de type personne */

- L'unification devient ainsi le calcul du GLB de deux symboles relative à l'ordre '<|'.

- Par une étape d'unification (plutôt que résolution), on obtient les réponses précédentes.

- Soit :

Cette déclaration est appelée une spécification de la *signature*.

L'ordre entre 'personne' et 'étudiant' dénote une imbrication de classes.

- Ceci permet de mettre en évidence les relations et améliorer les performances surtout quand la chaîne d'héritage augmente.

- Le gain vient de la différence des mécanismes d'unification (simple) et de la résolution (plus complexe) avec tous les changements de contextes, sauvegarde, ...

Exemple4 :

Un t_1 est un t_2;
Un t_2 est un t_3 , ..., un t_{n-1} est un t_n ;
t est un t_1;
t a la propriété *prop*.

➠ En Prolog :

$t_n(X)$:- t_{n-1} (X)

....

t_2 (X) :- t_1 (X).

$t_1(t)$.

prop (t).

Trouver un t_n avec la propriété *prop* demande n étapes de résolution;

On peut écrire :

$t <| t_1$.

$t_1 <| t_2$.

$t_{n-1} <| t_n$.

avec le fait

prop(t).

et la question

prop (X : t_n) ?

réussit avec une seule résolution.

LIFE

- Logic, Inheritance, Function, Equation (DEC).

- Intégration multi-paradigme

- Idée force : calculer sur des **Ψ-termes**, une généralisation des termes du 1er ordre

- Une généralisation de Prolog :
  la plupart des programmes Prolog "tournent" sous LIFE.

- Syntaxe très proche de celle de Prolog

- Possibilité d'interrogation incrémentale
  Un but est suivi de '**?**' Une règle est suiviz de '**.**'

  S'il y a une solution

  RC abandon de la question et retour au niveau précédent

  ; forcer un retour arrière et chercher une autre solution

  . revenir à l'interpète

Exemple :

```
pere(john, harry).
pere(john, mike).
pere(jharry, michael).
grand_pere(X, Y) :- pere(X, Z), pere(Z, Y).
```

Question :

```
grand_pere(A,B)?
```

*** Yes
A = john, B = michael.
--1> **pere(A,C)?** /* on repart avec la substitution */
*** Yes /* courante au lieu de {} */
A = john, B = michael, C = harry.
----2> **;**
*** Yes
A = john, B = michael, C = mike.
----2> **;**
*** No
A = john, B = michael.
--1> **pere(C, B)?**
*** Yes
A = john, B = michael, C = harry.
----2> **pere(A,C)?**
*** Yes
A = john, B = michael, C = harry.
------3>
*** No
A = john, B = michael, C = harry.
----2> **.**
>

Des termes Prolog aux Ψ-termes

- Structure d'enregistrement vs. expression fonctionnelle

  f(t1,..., tn) de Prolog est une structure d'enregistrement (arbre).
  Ce terme Prolog est une définition de type.
  L'unification de Prolog est le calcul d'une intersection de type.

- *personne (x, y, z)* est un record à 3 champs auquel le programmeur donne une interprétation (par exemple, x est le nom, y date de naissance, z le sexe).
  La sémantique opérationnelle implicite de ces constructeurs : ils dénotent l'ensemble (le type) de toutes les personnes dans la base.

- L'unification de Prolog est un cas trivial d'héritage :
  personne (x, y, z) est une notation générique de l'ensemble de toutes les personnes et *personne(nom(jean, X), Y, mâle)* dénote tous les mâles dont le prénom est Jean.

- Dans cette interprétation, l'unification agit comme un calcul de l'intersection de types.

  Par exemple, l'unification de
  personne(nom (X, X), Y, Z) et
  personne (nom (jean, X), Y, mâle)

  dénote l'intersection de l'ensemble des personnes dont le nom et le prénom sont identiques avec les personnes mâles dont le nom est Jean.
  Ce qui donne l'ensemble :
  personne (nom (jean, jean), Y, mâle)

- Considérer les termes de 1er ordre (purement syntaxiques) comme des constructeur avec arité fixe de la signature est insuffisant et inefficace.
  Par exemple, l'ajout d'un quatrième champ (N° sécu) à *personne* va remettre en cause tout ce qui est fait.

- Une autre insuffisance : dans *personne*, le premier argument doit toujours être le nom, il vaut mieux disposer d'un mécanisme de record où on peut nommer les champs sans ordre précis.

- Une autre limitation est d'avoir 'personne' comme racine du constructeur 'personne' qui filtre tout autre constructeur. On peut envisager que 'étudiant' soit sous-type de 'personne'. Un filtrage plus libre et gradué doit pouvoir être exprimé par une relation d'ordre (subsumption de type) sur la signature du constructeur.

- Bilan des limitations de Prolog:

  Termes fonctionnels non interprétés (ce sont des arbres)
  Arité fixe des termes composés
  Interprétation des positions des arguments nécessaire

  ➥ Interprétation de pere(x,y) ? x pere de y ?
  pere(x, y) --> pere(x, y, z) ?

  Les foncteurs sont des constantes et jouent le rôle de filtre d'instanciation.

- Idée : pour quoi ne pas poursuivre l'instanciation de
  personne(x, y) par etudiant(x, y) ?

  ➥ subsomption de types
  ➥ ordre partiel sur les constructeurs de types où l'unification devient le calcul d'une plus grande borne inférieure (GLB).

- Statut des variables :
  En prolog : joker (filtre lâche) ou étiquette imposant des contraintes d'unification.

  personne(_, _, homme)
  personne(nom(john, 1), A, homme).

  ➥ On peut étendre leur rôle : les variables peuvent figurer où un terme peut apparaître

➥ Unification vs. Résolution ==> LOGIN

Exemple :

```
> bob <| etudiant.          /* etudiant(bob) */
> etudiant <| homme.        /* homme(X) :- etudiant(X) */
> gentil(homme).            /* gentil(X) :- homme(X) */

> gentil(bob) ?
*** yes
> gentil(X) ?
*** Yes
X = homme.
--1> ;
*** No
```

Notion de Ψ-terme :

423	int	-5.56
real	"une chaine"	string
bob	[a,b,c,d]	

date(mercredi, 15, decembre)
date(1 => mercredi, 2 => 15, 3 => decembre)
[une, liste]
cons(celle_la, cons(aussi, []))

- terme Prolog : foncteur(t1, ... tn)
  Ψ-terme : foncteur(1 => t1, ..., tn)

- Les sortes (types) : constructeurs des données Life.

- Les sorts sont partiellement ordonnées dans une hiérarchie de sortes (relation notée >|)

  @ est la sorte la plus générale (top element)
  {} est la plus petite sorte (bottom element)

- Les sortes sont des valeurs comme les autres

```
moto <| deux_roues.
camion  <| quatre_roues.
voiture <| quatre_roues.
moto <| vehicule.
camion <| vehicule.
voiture <| vehicule.
```

- Les variables peuvent apparaître partout dans un Ψ-terme

- Etiquette et co-référence (éventuellement cyclique)

  x : t. le Ψ-term t est étiqueté par x
  x x : @

  x : t1 & t2. x=t1, x=t2.

  write(vehicule & quatre_roues) ?
  -> écriture de 'voiture' puis de de 'camion' (retour arrière)

  X : personne(id => N : nom,
  ne => date,
  fils => personne (id => N,
  pere => X)).

  pere(fils(X)) = X
  id(fils(X)) = id(X)

- {t1; t2; ...; tn} est un terme disjonctif.

A={1;2;3}? ==> A=1; A=2; A=3
note := {bonne; mauvaise}. ==> note = bonne; note = mauvaise
p({1; 2}). ==> p(1). p(2).

- L'intersection de deux types est appelée GLB (Greatest Lower Bound). Lorsque cette intersection n'est pas unique, on construit l'union des "Maximal Common Subsorts" des deux types.

peugeot_306 <| voiture.
deux_roues ∩ vehicule = moto
quatre_roues ∩ vehicule = {voiture; camion}
deux_roues ∩ quatre_roues = {}
peugeot_306 ∩ voiture = peugeot_306
camion ∩ @ = camion.

> camion <| vehicule.
*** yes
> mobile(vehicule).
*** yes
util(camion).
*** yes
mobile(X), util(X) ?
*** yes
X = camion

-
int <| real.
GLB(list, real) = {}

Unification des Ψ-termes

- U = @ , V : @ , U = V ?

==> U=@, V=U

- U = int , V = 5.0 , U=V?
  ==> U=5, V=U

- U = int , V = 5.6 , U=V?
  ==> U=5, V=U

- U = [A, B, C], V = [1, 2, 3] , U = V ?
  ==> A = 1, B=2, C=3, U =[A, B, C], V = U

- U = [H | T], V = [a, b, c, d], U = V ?
  ==> H = a, T=[b,c,d], U=[H|T], V=U

Exemples plus complexes :

- Chaque label de GLB est dans l'intersection des constructeurs et les sous termes sont l'union des sous termes.

- U = voiture(couleur => rouge, roues => 4),
  V = vehicule(roues => N : int) , U = V ?
  ==> N = 4,
  U = voiture(couleur => rouge, roues => N),
  V = U

étudiant <| personne.
employé <| personne.

bob <| étudiant.
piotr <| étudiant.
pablo <| étudiant.

elena <| survaillant.
simon <| survaillant.

administratif <| employé.
enseignant <| employé.
surveillant <| étudiant.
surveillant <| administratif.

don <| enseignant.
john <| enseignant.
sheila <| enseignant.

art <| administratif.
judy <| administratif.

```
X = etudiant
            (co-chambre => employé(groupe => S),
             conseiller => don(secretaire => S)).

Y = employe
            (conseiller => don(assistant => A),
             co-chambre => S : etudiant(groupe => S),
             aide => simon(épouse => A)).

X = Y ?

      X = surveillant
            (conseiller => don(assistant => A,
                               secretaire => B : surveillant(groupe => B)),
             aide => simon(epouse => A),
             co-chambre => B),
      Y = X.
```

Différences avec Prolog

En Prolog

- La signature est un treillis plat (tous les symboles autre que "top" et "bottom" sont incomparables).
- Les étiquettes peuvent apparaître seulement aux feuilles et sont alors de type @.
- Les étiquettes des attributs sont des séquences d'entiers de longueur fixée (l'arité) pour chaque symbole de la signature.

- GLB obtenu par l'unification du première ordre;
- LUB obtenu par l'anti unification (généralisation du premier ordre).

Fonctions (FOOL & Le FUN)

- Règles de réécriture transformant des Ψ-termes en des Ψ-termes.
- Utilisation d'un filtrage plutôt que d'une unification sur les arguments des fonctions.

```
> fact(0) -> 1.
> fact(N : int) -> N * fact(N-1).
```

```
> write(fact(5)) ?
120
*** yes
```

```
somme([]) -> 0.
somme([X|Y]) -> X+somme(Y).
```

```
> A=somme([1,3])?
*** Yes
A = 4.
--1> ;
*** No
```

- Les entêtes des fonctions n'ont pas le droit d'unifier (en fait, on unifie puis on défait l'unification).

Avec f(X, X) -> 1.
et l'appel A=f(X,Y), X et Y ne sont pas rendus identiques.

Résiduation

```
> A = fact(B)?
*** yes
A = @, B = @~
--1> B= real ?
*** yes
A = @, B = real~          /* on a int <l real */
----2 > B=5 ?
*** yes                   /* 5 <lint, fact(B:5) s'évalue */
A=120, B=5
------3 >
*** No
----2 > A=123?
*** yes
A = 123, B = real~
------3 > B=6 ?
*** No
A = 123, B = real~
------ 3>
```

- Les fonctions sont déterministes :
  avec f(X, X) -> ...
  l'appel f(a,b) ne fera rien si a et b ne sont pas unifiables
  résiduation s'ils sont unifiables
  évaluation si a et b sont unifiés dans le contexte

Currification : application partielle de fonctions

- La currification est l'action d'obtenir de
  f(x, y) la fonction (f(x)) (y).
  f : x × y -> ...

  f currifiée :
  f : x -> g(y)

g : y -> ....

- Le résultat de la currification est une fonction
- La forme currifiée de f(a => X, b=> Y) est

f(a => X) & @ (b => Y)

ou f(b => Y) & @(a => X)

- > f(X, Y, Z) -> [X, Y, Z].

*** yes

> A = f(a, 3 => c) ? /* f appelée avec 2 paramètres */

*** yes

A = f(a, 3 => c) .

--1 > A = f(2 => b) ?

*** yes

A = [a, b, c].

Variables fonctionnelles

```
map(F, []) -> [].
map(F, [H | T]) -> F(H) | map(F, T)].
```

> L = M(F, [1, 2, 3, 4]) ?

*** Yes

F = @, L = @, M = @~

--1 > M = map ?

*** Yes

F = @~~~~, L=[@, @, @, @], M=map.

----2> F = +(2 => 1) ?

*** Yes

F = +(2 => 1) , L=[2, 3, 4, 5], M=map.

------3>

- La résiduation, la currification et les fonctions variables donne une flexibilté grande aux foctions :

-

```
quadruple -> *(2 => 4).
pick_arg({5; 3; 7}).
pick_func({quadruple, fact}).
```

```
test :-   R=F(A),
          pick_arg(A),
          pick_func(F),
          write(F, "appliquée à", A, "donne :", R),
          nl, fail.
```

```
> test?.
*(2=>4) appliquée à 5 donne 20
fact appliquée à 5 donne 120
*(2=>4) appliquée à 3 donne 12
fact appliquée à 3 donne 6
*(2=>4) appliquée à 7 donne 28
fact appliquée à 7 donne 5040
*** No
```

Quote & Eval : Echappement à l'évaluation

```
> X = 1 +2 ?
*** yes
X=3
--1 > Y=`(1+2)?
*** yes
X=3, Y=1+2
----2> Z=eval(Y)?
***
X=3, Y=1+2, Z=3
```

Contraintes sur les types (démons)

- On peut attacher des propriétés aux types : attributs, relations ou fonctions.

- Ces propriétés seront vérifiées pendant l'exécution. Elles sont héritées par les sous-types.

- On fait précéder la définition de '::'

```
> :: personne(age => int).          /* le champs age est contraint */
*** yes                            /* à être un entier */
```

```
> homme <| personne.
*** yes
> A=homme?
*** Yes
A= homme(age => int).
--1>
```

- ```
  :: vehicule( marque => string,
               nombre_de_roues => int).

  :: voiture(nombre_de_roues => 4).
  voiture <|vehicule.

  >A=voiture?
  *** Yes
  A=voiture( marque => string,
             nombre_de_roues => 4).
  --1>
  ```

Notations

- homme := personne(genre => masculin).

  est équivalent à :

  ```
  homme <| personne.
  :: homme(genre => masculin).
  ```

- arbre := {feuille ; noeud(gauche=> arbre, droit => arbre)}

  est équivalent à :

  ```
  feuille <| arbre.
  noeud <| arbre.
  :: noeud(gauche => arbre, droit => arbre).
  ```

-
  ```

```
:: rectangle( longeur => L : real,
              largeur => S : real,
              aire => L * S).
carre := rectangle( cote => S,
                    longeur => S,
                    largeur => S).

> R = rectangle(aire => 16, largeur => 4) ?
*** Yes
R= rectangle( cote => 16,
              longeur => 4,
              largeur => 4).

--1> R=carre ?
*** Yes
R = carre(aire => 16,
              longeur => A : 4,
              largeur => A,
              cote => A).
----2>
```

Démons

-

```
:: devoue(religion => F, prie => X) | figure_sainte(F,X).
figure_sainte(musulman, allah).
figure_sainte(juif, yahve).
figure_sainte(chretien, jesus).
```

```
> Z=devoue(religion => X, Y)?
*** Yes
X = musulman, Y = @, Z = devoue(Y,prie => allah,religion => X).
--1> ;
*** Yes
X = juif, Y = @, Z = devoue(Y,prie => yahve,religion => X).
--1> ;
*** Yes
X = chretien, Y = @, Z = devoue(Y,prie => jesus,religion => X).
--1> ;
*** No
```

-

```
> :: I : int | write(I, " ").   /* à chaque instance d'entier, on affiche */
*** Yes
> A = 5 * 7 ?
5 7 3 5
*** Yes
A= 35.
--1 > B=fact(5) ?
5 1 4 1 3 1 2 1 1 1 0 1 1 2 6 24 120
*** Yes
A = 35, B=120
----2>
```

-

```
> :: C : cons | write(C.1), nl.
*** yes
> A=[a,b,c,d] ?
d
c
b
a
*** Yes
A=[a,b,c,d].
```

Types récursives

```
list := {[]; [@|list]}.
```

Programmation O-O avec Life

- Définition de classes par la définition de types :

```
:: classe(  champs1 => valeur1,
            champs2 => valeur2,
            ...).
```

- Pour dire que *classe1* hérite des propriétés de la *classe2* :

  classe1 <| classe2.

- Instances crées dans les programmes :

  >X=int ? /* crée une instance de la classe int */
  > X=int, Y=int ? /* 2 instances *différentes nouvelles* créées */

Avec :
  > X=23, Y=23 ? /* deux instances fraîches de la classe 23 */
Et
  >f(A, A) -> 1 .

  f(X, Y) ne déclenche pas f; pas d'unification car X et Y sont des instances différéntes

Il faut rendre X et Y identiques par '='
> X=23, Y=23, Z=f(X,Y)?
*** Yes
X = 23~, Y = 23~, Z = @.
--1> X=Y?
*** Yes
X = 23, Y = X, Z = 1.
----2>
*** No

- Life : langage de programmation expérimental
  Compilateur en cours de développement
  WildLife diffusé gratuitement par DEC

- Applications
  - Linguistique
  - Graphique sous contraintes
  - Systèmes experts

Exemples

Les n-reines :

```
queens (N : {n;int}, Rep) :-
        place_queens ([], interval(1,N), interval(1,N), Rep).

place_queens (Queens, [], [], Queens).
place_queens (Queens, [X|Rows], Cols,Rep) :-
      choose_one (Y, from => Cols, rest => RestCols),
      check_diagonals (X, Y, Queens),
      place_queens ([(X,Y)|Queens], Rows, RestCols, Rep).

check_diagonals (X, Y, []) :- !.
check_diagonals (X, Y, [(X1,Y1)|Queens]) :-
      X+Y =\= X1+Y1,
      X-Y =\= X1-Y1,
      check_diagonals (X, Y, Queens).

interval (From, To) ->
      cond (From < To,
          append (interval(From+1,To-1), [From,To]),
          cond (From =:= To, [From], [])).

choose_one (from => []) :- !, fail.
choose_one (H, from => [H|T], rest => T).
choose_one (X, from => [H|T], rest => [H|L]) :-
      choose_one (X, from => T, rest => L).
```

queens(4)?
===> solutions

Les nombres premiers

A prime number is a positive integer whose number of proper factors is exactly one.

```
prime :=  P:posint | number_of_factors(P) = one.

posint := I:int | I>0.
```

```
number_of_factors(N) -> cond (N<2, undefined,
                                        factors_from(N,2)).

factors_from(N:int,P:int) ->
      cond(P*P>N,
          one,
          cond(R:(N/P)=:=floor(R),
              many,
              factors_from(N,P+1)
          )
      ).

primes_to(N:int) :-
      write(posint_stream_to(N):prime),
      nl,
      fail.

primes_to_gc(N:int) :-
      write(posint_stream_to(N):prime),
      nl,
      gc,
      fail.

posint_stream_to(N:int) ->
      cond(N<1,
          undefined,
          {1;1+posint_stream_to(N-1)}
      ).

all_primes :-
      write(posint_stream:prime),
      nl,
      fail.

posint_stream -> {2;1+posint_stream}.
```

Question :
> primes_to(20)?

2:prime
3:prime
5:prime
7:prime
11:prime
13:prime
17:prime
19:prime

PERT

Les contraintes sont installées pendant les vérifications de types. Elles sont relâchées dès qu'assez d'information est connue.
Idem pour les activités.
Ainsi, tout calcul n'est pait qu'une seule fois

```
op(400,yfx,##)?
X##Y -> project(Y,X).

%%%% Do the scheduling %%%%
:: A:activity( duration =>D:real,
              earlyStart => earlyCalc(R),
              lateStart  => {1e500;real},
              prerequisites => R:{[];list} ) I lateCalc(A,R).

% Pass 1: Calculate the earliest time that A can start.
earlyCalc([]) -> 0.
earlyCalc([BIListOfActs]) ->
       max(B##earlyStart+B##duration,earlyCalc(ListOfActs)).

% Pass 2: Calculate the latest time that A's prerequisites can start
% and still finish before A starts.
lateCalc(A,[]) -> succeed.
lateCalc(A,[B:activityIListOfActs]) ->
       lateCalc(A,ListOfActs) I
       assign(LSB:(B##lateStart),
                     min(LSB, A##earlyStart-B##duration)).

% Wait until B is an integer before doing the assignment:
```

```
assign(A,B:int) -> succeed | A<-B.

%%%% the output %%%%
visAllActs([]).
visAllActs([A|B],N:{1;real}) :- visualize(A,N), visAllActs(B,N+1).

visualize(A:activity,N:int) :-
        write("Activity ",N,": " ,A##earlyStart,"      ", A##earlyStart ), nl.
```

Question :

```
A1=activity(duration=>10),
A2=activity(duration=>20),
A3=activity(duration=>30),
A4=activity(duration=>18,prerequisites=>[A1,A2]),
A5=activity(duration=>8 ,prerequisites=>[A2,A3]),
A6=activity(duration=>3 ,prerequisites=>[A1,A4]),
A7=activity(duration=>4 ,prerequisites=>[A5,A6]),
visAllActs([A1,A2,A3,A4,A5,A6,A7]) ?
```

```
Activity 1: 0          0
Activity 2: 0          0
Activity 3: 0          0
Activity 4: 20  20
Activity 5: 30         30
Activity 6: 38         38
Activity 7: 41         41
```

Appendixe

La mise en oeuvre : extension des termes de 1er ordre

- Interprétation opérationnelle des termes :
  un littéral $P(t_1, ---, t_n)$ avec P : symbole prédicatif,
  t_i : termes (fonctionnels) de 1er ordre.

Soit $\{ \Sigma_n \mid n \in N \}$ un ensemble disjoint de signature indicées de symboles fonctionnels

La signature $\Sigma = \cup_{n \in N} \Sigma_n$ contient les symboles devant être interprétés dans un modèle sémantique donné où n est l'arité (nombre d'arguments) des symboles fonctionnels. Les symboles d'arité 0 dénotent les constantes (Σ_0 n'est pas vide pour obtenir des modèles non vides).

Soit V un ensemble infini énumérables de variables.

- L'ensemble $T_{\Sigma,V}$ des termes de 1er ordre de la signature Σ et les variables V est définit inductivement par :
  - les éléments de V et de Σ_0 sont des termes de 1er ordre,
  - si $f \in \Sigma_n$ et $t_1, \ldots, t_n$ sont des termes de 1er ordre, alors $f(t_1,\ldots, t_n)$ est un terme de 1er ordre.

  ➥ Unification vue comme le calcul de GLB.

- On augmente la signature du symbole $\perp$ (échec).
- Une variable est une étiquette avec une contrainte d'égalité (toute occurrance d'une variable à la même valeur).
- Toute occurrence d'une variable représente le même terme.
- On augmente la signature du symbole T (top) comme le plus haut type; c'est l'élément (le type) le plus grand.

- Les variables peuvent figurer comme étiquette n'importe où dans les termes et imposent une contrainte d'égalité.

Calcul des structures de type partiellement ordonnées

- Ψ-terme : un type structure :
  1) Une racine
  2) Un label d'attribut = les symboles de champs de record.
     Un label est une fonction de la racine vers le type du sous-terme associé.
     La concaténation des labels : composition de fonction notée :
     (id => (nom => ---
  3) Les contraintes de coréférence (par exemple : les variables).
     Une coréférence spécifie qu'un diagramme fonctionnel d'attributs est commutatif.

Exemples de Ψ-termes :

1) personne (id ⇒ nom ,
   naissance ⇒ date (jour ⇒ int ,
   mois ⇒ string ,
   an ⇒ int)
   père ⇒ personne)

   Racine : *personne* avec 3 sous termes sous les labels
   Labels : id , naiss , père ;
   personne, date : constructeurs

2) personne (id ⇒ nom (prénom ⇒ string ,
   famille ⇒ X : string) ,
   père ⇒ personne (id ⇒ nom (famille ⇒ X : string)))

L'étiquette X représente une contrainte de coréférence
==> des sous-structures identiques.

Les référence X : ... doivent être identiques.
Une seule référence X : string suffit.

3) personne (id ⇒ nom (prénom ⇒ string ,
famille ⇒ string ,
père ⇒ X : personne (fils ⇒ personne (père ⇒ X)))

La référence cyclique étiquetté pas X.

- La signature de type est un ensemble partiellement ordonné de symboles. La signature est augmentée de T et de ⊥. Les symboles de type représentent des ensembles d'objets et l'ordre partiel sur la signature est l'inclusion. T représente l'ensemble de tous les objets = univers. On omet d'écrire T et l'on le considère implicite pour tout type :

personne (id ⇒ (prénom ⇒ string ,
famille ⇒ X),
père ⇒ personne (id ⇒ nom (famille ⇒ X))).

Le type T est le type de {id, id.famille, père.id.famille}

- ⊥ est l'ensemble vide, il est le type d'aucun objet. Toute occurrence de ⊥ dans un terme rend le terme = ⊥. Un terme bien formé t_1 est sous-type d'un autre terme bien formé t_2 si :

- la racine de t_1 est sous-type de la racine de t_2
- tous les labels d'attributs de t_2 sont labels de t_1 et leur sous termes respectifs sont sous types.
- toutes les liens de contraintes de coréférence de t_2 doivent être liés dans t_1

Exemple5

étudiant <| personne
paris <| nom_ville
étudiant (id ⇒ nom (prénom ⇒ string ,
famille ⇒ X : string) ,
vit_a ⇒ Y : adresse (ville ⇒ paris) ;
père ⇒ personne (id ⇒ nom (famille ⇒ X) ;
vit_a ⇒ X))

est sous-type de :

personne (id ⇒ nom (famille ⇒ X : string) ,
vit_a ⇒ adresse (ville ⇒ nom_ville),
père ⇒ personne (id ⇒ nom (famille ⇒ X))).

- Si, dans une signature Σ, les GLB et LUB existent pour toute paire de symboles de type (étant donné l'ordre dans la signature) alors GLB et LUB existent pour l'ordre sur les termes bien formés. Autrement dit, l'ordre des termes bien formés étend un (semi) treillis de la signature aux termes.

Exemple6
Avec :

- Le LUB de
enfant (connait ⇒ X : personne (connait ⇒ reine ,
déteste ⇒ Y : roi) ,
déteste ⇒ enfant (connait ⇒ Y,
aime ⇒ marraine),
aime ⇒ X)

et de :
adulte (connait ⇒ adulte (connait ⇒ fée),

déteste ⇒ personne (connait ⇒ X : roi ,
aime ⇒ X)).

est :

personne (connait ⇒ personne ;
déteste ⇒ personne (connait ⇒ roi ,
aime ⇒ roi)).

Où les labels de LUB est le terme généralisant les constructeurs et les sous termes sont l'intersection des sous termes.

et leur GLB est :

teen-ager (connait ⇒ X : adulte (connait ⇒ marraine ,
déteste ⇒ X : marraine),
déteste ⇒ enfant (connait ⇒Y,
aime ⇒ Y),
aime ⇒ X).

Où les labels de GLB est l'intersection des constructeurs et les sous termes sont l'union des sous termes.

- On constate que l'unification de Prolog est un cas restrictif de ci-dessus. Les termes de 1er ordre sont des Ψ-termes tels que :

1) la signature est un treillis plat, tout symbole (de type, exceptés T et ⊥) sont incompatibles.

2) les étiquettes (les variables) ne figurent qu'en feuille (avec le symbole T dans ce cas)

3) les labels sont une séquence fixe de nombre naturels our chaque symbole de la signature.

De plus, le GLB est l'unification et le LUB est la généralisation de 1er ordre.

- Le terme du 1er ordre $f(t_1, ... , t_n)$
  est le Ψ-terme $f(1 \Rightarrow t_1 , ..., n \Rightarrow t_n)$

- Pour intégrer l'héritage dans la programmation logique, on doit définir l'unificateur de Ψ-termes.

  On suppose que la signature et le semi-treillis inférieur, c'est-à-dire : le GLB existe pour toute paire de symbole de type.

  L'algorithme d'unification doit calculer, pour une paire de termes, le GLB qui est sous-type des deux termes.

Algorithme de principe

- L'unification de deux sous-termes consiste à effectuer l'intersection des types et l'union des labels.

Soit deux termes :

$$s = X_O : f (\quad k_1 \Rightarrow X_1 : s_1,$$
$$\quad .....$$
$$\quad k_m \Rightarrow X_m : s_m) \quad \text{et}$$

$$t = Y_0 : g (\quad l_1 \Rightarrow Y_1 : t_1,$$
$$\quad .....$$
$$\quad l_n \Rightarrow Y_n : t_n)$$

Soit **var(t)** l'ensemble des étiquettes du Ψ-terme t. Les étiquettes $X_0 ... X_m$ et $Y_0 ... Y_n$ sont mises en place même si elles ne figurent pas dans les termes initiaux car, on peut rajouter une nouvelle étiquette X_i (ou Y_i) si X_i (Y_i) $\notin$ var(s) $\cup$ var(t).

Avant d'effectuer l'unification, on s'assure que les ensemble var(s) et var(t) sont disjoints. ($var(s) \cap var(t) = \emptyset$).

Si ça n'est pas le cas, on renomme les variables (les étiquettes).

L'algorithme d'unification opérationnelle informalle

(Calcul de l'instance commune)

L'algorithme d'unification calcule le GLB de deux termes s et t. Le terme résultant (instance) est sous-terme des termes unifiés s et t.

1) Pour tout couple d'étiquette $< X_i, Y_j >$,
S'il existe une étiquette $Z_p = \{X_i, Y_j\}$ représentative d'un ensemble E d'étiquette où figure X_i et/ou Y_j alors $E = E \cup \{X_i, Y_j\}$ et le type représentatif de Z_p sera :

$type(Z_p) \cap type(X_i)$ si $X_i \in Z_p$
$type(Z_p) \cap type(Y_j)$ si $Y_j \in Z_p$

$Z_\emptyset$ sera construit au départ par $Z_\emptyset = \{X_0, Y_0\}$ dont le type est $type(X_0) \cap type(Y_0)$. Ce type sera le type principale du terme final.

2) Pour tout constructeur de type $\alpha = type(X_i) \cap type(Y_j)$ dont on vient de définir le type, faire l'union des labels de X_i et de X_j. Cette union donne l'ensemble des labels du constructeur α dans le terme final.

Exemple :

X_0 : etudiant(conseiller => X1 : faculté(secrétaire => X3 : admin,
assistant => X0),
co-chambre => X2 : employé(representant=> X3))

Y_0 : employe(conseiller => Y1 : f1 (secrétaire => Y4 : employé,
assistant => Y5 : personne),
co-chambre => Y2 : etudiant(representant => Y2),
aide => Y3 : w1(epouse => Y5)).

Avec

On obtient

Z_0 : bureau(conseiller => Z1 : f1(secrétaire => Z2,
assistant => Z0),
co-chambre => Z2 : bureau(représentant => Z2),
aide => Z3 : w1(epouse => Z0)).

Détails (suivre l'ordre ci-dessous):

1 - type(X0) $\cap$ type(Y0) = bureau. Les labels de Z0 = $\cup$ des labels de X0 et de Y0.

2- On unifie X0 et Y5. X0 est déjà dans Z0 dont le type actuel est bureau. On augmente Z0 de Y5 et on calcule {bureau} $\cap${personne} = {bureau}. bureau sera le type de Z0 qui sera le type de X0, Y0 et Y5.

3- Il faut unifier X3 et Y2. On a X3$\in$Z4:admin, Y2$\in$Z2 : bureau. On calcule : Z5 : bureau={X2,Y2, X3, Y4}.
Le type est {admin $_{Z4}$} $\cap${bureau$_{Z2}$} = {bureau}. bureau sera le type de tous les éléments de Z5.

4- On cherche Y5. Y5$\in$Z0 : bureau ==> Z0 ={X0, Y0, Y5}. Z0 ne se modifie pas et Y5 prend le type de Z0, i.e. bureau.

Idem pour Z6=Z0:bureau.

Le terme final est le résultat du parcours en profondeur de cet arbre en traitant les labels. Si l'on fait une lecture par niveau, seules les premières occurrences des variables rencontrés changent.

Un exemple de résolution :

On veut exprimer le faits suivants :

- un étudiant est une personne
- Peter, Paul & Marie sont des étidiants
- bonne et mauvais sont des notes
- une bonne note est une bonne chose
- 'A' et 'B' sont des bonnes notes
- 'C', 'D' et 'F' sont des mauvais notes.

On a :

L'expression en LOGIN donne :

```
etudiant <| personne.
{peter; paul; marie} < etudiant.          /* noté etudiant := {....} */
{bonnenote; mauvaisnote} <| note.
bonnenote <| bonnechose.
{a; b} <| bonnenote.
{c; d; f} <| mauvaisenote.
```

De plus :

```
aime(X : personne, X).
aime(peter, marie).
aime(personne, bonnechose).

obtient(peter, c).
obtient(paul, f).
obtient(marie, a).
```

Et toute personne est contente si

- elle aime quelque chose qu'elle obtinet ou
- elle aime un/une autre (Y) et Y obtint une bonne chose.

(1) content(X : personne) :- aime(X, Y), obtient(X, Y).
(2) content(X : personne) :- aime(X, Y), obtient(Y, bonnechose).

Question :

?- content(X).

==> X = marie par (1) : car elle a obtenu une bonne chose (note)

Retour arrière

==> X = marie car elle s'aime et elle a obtenu une bonne chose.

Retour arrière

==> X = peter car il aime marie qui a obtenu une bonne chose.

Conclusion

LOGIN est simplement Prolog où les termes du premier ordre sont remplacés par des Ψ-termes (l'unification des Ψ-termes remplace celle des termes du 1er ordre).

LOGIN est bien adapté aux paradigmes objets en programmation logique de même que dans les applications en BD.

Extension Objet de Prolog : L & O

objet : label(identifiant)
+ programme Prolog

Définition :

```
Label : {  axiome1.
           axiome2.
           ....
        }
```

```
train(Vitesse, Couleur, Region, Terrain) : {
    couleur(Couleur).
    vitesse(Vitesse).
    region(Region).
    terrain_adequat(Terrain).
    temps_voyage(Dist, T) :- T = Dist / S
        }.
```

Une instance de train :

```
    train(160, vetre, bretagne , plat) : {
    couleur(verte).
    vitesse(160).
    region(bretagne).
    terrain_adequat(plat).
    temps_voyage(Dist, T) :- T = Dist / 160
        }.
```

- Possibilté d'utiliser des variables comme label :

Avec

```
    bretagne : {
            ...
        type_de_terrain(plat)
            }.

    compatible(Train, Pays) :-
        Pays : type_de_terrain(T), Train : terrain_adequat(T).
```

Question :

compatible(tgv, isere)?

• Définition de relations :

```
personne(Sex, Age) : {
        ...
        age = Age.
        aime(X) :- X:age(A), A < 50.
        aime(X) :- X:aime(self)
            }.
```

Question :

jean : aome(marie)?

.....