Prolog
et
Traitement Automatique des Langues

Éric de la Clergerie
Eric.De_La_Clergerie@inria.fr

ALPAGE – INRIA
http://alpage.inria.fr

Cours M2 LI 2008

Aujourd'hui: Représenter l'information

Cinquième partie V

Représenter l'information

Motivations

La notation par termes Prolog n'est pas toujours adaptée à l'écriture compacte

- de termes riches en information
- mais en général partiellement spécifiés dans les productions.

Exemple d'un verbe avec ses diverses informations morpho-syntactiques :
v(Number,Gender,Person,Tense,Mood,Aux,Voice)

Facile de se tromper :

- sur l'arité : v(Number,Gender,Person,Mood,Aux,Voice)
- sur l'ordre des arguments : v(Number,Gender,Person,Mood,Tense,Aux,Voice)
- sur les valeurs des propriétés :
  v(pl ,Gender,Person,pres,ind,Aux,Voice)
  v(plural ,Gender,Person,present,ind,Aux,Voice)

Notation vite lourde car tout les champs doivent être explicitement remplis, même si c'est de manière anonyme : v(_,_,3,_,_,_,_)

Mise à jour compliquée si on change l'arité d'un terme :
v(Number,Gender,Person,Tense,Mood,Aux,Voice,SupportVerb)

Motivations (suite)

On recherche des notations :

- évitant les problèmes d'arité et d'ordre sur les propriétés
- permettant de ne fournir que les informations utiles (sous-spécification)
- permettant l'ajout de nouvelles propriétés
- évitant au maximum les erreurs en spécifiant des contraintes de bonne formation
  similaire au typage en ML (CAML) ou aux DTD/schémas XML en XML
- exploitant ces contraintes pour encore plus de compacité
- exploitant ces contraintes pour plus d'efficacité pendant l'exécution

Quelques pistes

Abstraitement, un objet linguistique

- d'un certain type
- se caractérise par un ensemble de propriétés ou traits
- prenant leurs valeurs dans des domaines de valeurs, souvent finis

Les verbes de type v, avec les propriétés et valeurs

propriétés	valeurs
number	sg, pl
gender	fem, masc
person	1,2,3
tense	present, future, imperfect, past,perfect, . . .
mood	indicative, subjonctive, conditional, infinitive, . . .
aux	être, avoir
voice	active, passive

Proposition :

- domaines finis de valeurs
- structures de traits, typés ou non

Préliminaire : unification immédiate

En DyALog, l'opérateur :: permet l'unification de termes pendant la phase de lecture des programmes (plutôt que pendant la compilation ou l'exécution).

```
X::gn(Number,Gender) --> X, rel(gap(X)).
```

(presque) équivalent à :

```
gn(Number,Gender) --> gn(Number,Gender), rel(gap(Number,Gender)).
```

Motivation principale pour :: : partager des occurrences de termes complexes

Unification immédiate (suite)

Attention : Unification immédiate délicate d'utilisation :

```
p(X) :- q(X::f(a)).
p(X) :- r(X::g(b)).
```

$\Rightarrow$ OK

```
p(X) :- q(X::f(a)) ; r(X::g(b)).
```

$\Rightarrow$ Erreur ! Échec de l'unification immédiate

Important : Portée de l'unification immédiate $=$ portée des variables
$=$ "clause" Prolog (clause, fait, production, ...)

Outline

Principe

Possibilité de

1. nommer un ensemble fini de valeurs (atomiques)
2. utiliser des variables prenant leur valeur dans des sous-ensembles

```
%% Declaration
:-finite_set(mood,[cond,ger,imp,ind,inf,part,subj]).

%% Utilisation
s(Mood) --> gv(Mood::mood[inf,imp,part,ger]).
s(Mood) --> gn, gv(Mood::mood[cond,ind,subj]).
```

Spécifique DyALog, mais proche :

- énumérations à la C `enum` (ou en CAML)
- programmation logique avec contraintes sur les domaines finis cc(FD), Van Hentenryck, V. Saraswat, Y. Deville
  mais mise en oeuvre bien plus limitée dans DyALog (pas de contraintes)

Quelques extensions

- Toutes les valeurs : person[] $\equiv$ person[1,2,3]
- Définition par différence :

```
%% Utilisation
s(Mood) --> gv(Mood::mood[~ [cond,ind,subj]]).
s(Mood) --> gn, gv(Mood::mood[cond,ind,subj]).
```

- Définition de sous-ensembles :

```
%% Déclaration
:-subset(finite,mood[cond,ind,subj]).
:-subset(non_finite,mood[~ [cond,ind,subj]]).
%% Utilisation
s(Mood) --> gv(Mood::non_finite[]).
s(Mood) --> gn, gv(Mood::finite[]).
```

Note : Les sous-ensembles sont du *sucre syntaxique* :
finite [ind,subj] $\rightsquigarrow$ mood[ind,subj]

Aspects formels

Un domaine fini de type T est associé à un domaine fini de valeurs $\mathcal{D}_T$.

- L'interprétation de FD-terme t[a1 ,..., an] est $(T, \{a_1, \ldots, a_n\})$, avec $a_i \in \mathcal{D}_T$
  (cas spécial) $\mathcal{I}(\text{t}[]) = (T, \mathcal{D}_T)$
- L'interprétation de t[~[a1 ,..., an]] est $(T, \mathcal{D}_T / \{a_1, \ldots, a_n\})$
- L'unification de deux FD-termes $(T, \mathcal{D}_1)$ et $(T, \mathcal{D}_2)$ donne $(T, \mathcal{D}_1 \cap \mathcal{D}_2)$
  avec échec de l'unification si $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$
- le terme $(T, \mathcal{D}_1)$ généralise (ou subsume) le terme $(T, \mathcal{D}_2)$ ssi $\mathcal{D}_2 \subset \mathcal{D}_1$

Unification : exemple

```
?-X=mood[cond, ger, imp, ind, inf],
  Y = mood[ger, imp, ind, inf, part],
  X=Y.
%% Answer:
  X = A__12::mood[ger, imp, ind, inf]
  Y = A__12::mood[ger, imp, ind, inf]
```

Note : les deux variables *X* et *Y* ont été redirigées vers une nouvelle variable A__12, liée au nouveau domaine de valeurs.

```
?- finite[] = non_finite[].
%% => échec !
```

Singleton

DyALog propose un traitement spécial des ensembles singleton
le terme singleton $(T, \{a\})$ est considéré comme équivalent à la constante a.

```
?-X=mood[ind].
   X = ind
```

Impact sur l'unification :

- l'unification de 2 FD-termes peut être une constante

```
?-X=mood[ind,subj], X=mood[ind,cond].
          X = ind
```

- Un FD-terme peut être unifié avec une constante

```
?-X=ind,X=mood[ind,cond].
          X = ind
```

Singletons (suite)

Attention : L'interprétation des singletons est pratique
mais incorrecte (formellement) !
En effet, l'unification n'est plus associative : $(t_1 \sqcup (t_2 \sqcap t_3)) \neq ((t_1 \sqcup t_2) \sqcup t_3)$

```
:-finite_set(foo,[a,b,c]).
:-finite_set(bar,[c,d,e]).

?-X=foo[],X=bar[],X=c.
%% => Échec
?-X=foo[],X=c,X=bar[].
        X = c
```

Représentation et efficacité

Les FD-termes sont utiles d'un point de vue descriptif :

```
s(Mood) --> gn, gv(Mood::finite[]).
```

équivalent à :

```
s(Mood) --> gn, gv(Mood), { domain(Mood,[cond,ind,subj]) }.
```

Mais FD plus efficace en pratique car évitant une exploration en retour arrière des différentes valeurs.

Efficacité

L'efficacité provient d'une implantation par vecteurs de bits :

```
%%          bit pos:0 1 2 3 4
:-finite_set(foo,[a,b,c,d,e]).
?-X=foo[a,c,d],
  X =.. T.    %% destructure X en T
%% Answer
   X = B__5::foo[a, c, d]
   T = [$SET$,B__5,foo(a, b, c, d, e),13]  %% 13 = 1+4+8 = 10110
```

ie DVar::foo[a, c, d] représenté en interne par 'SET'(DVar,foo(a,b,c,d,e),13)

Unification $\equiv$ **et**-booléen $\&\& : (T, \mathrm{bv}_1) \sqcup (T, \mathrm{bv}_2) = (T, \mathrm{bv}_1 \&\& \mathrm{bv}_2)$

- pas de retours arrière
- unification en temps (quasi) constant

Idem pour le test de subsumption $t_1 \sqsubseteq t_2$ ssi $\mathrm{bv}_1 \&\& \mathrm{bv}_2 == \mathrm{bv}_2$

Termes déréférençables

Les FD-termes sont un cas particulier de termes déréférençables (DyALog) :

- en Prolog, une variable X peut être liée à un terme t
  t est la valeur de X par déréférencement
- en DyALog, un deref term t_1 peut être lié à un terme t_2
  t_2 est la valeur de t_1 par déréférencement

Les deref-termes possède une variable en 1er argument
'SET'(DVar,foo(a,b,c,d,e),13)

Un deref-term est lié quand cette variable est liée.

Un dérérencement peut suivre une chaîne de liaison
X::foo[a,b,c,d,e] $\rightsquigarrow$ (liaison X) Y::foo[a,b,c] $\rightsquigarrow$ (liaison Y) b

Quelques exemples

Très nombreux usages des domaines finis en TAL,

- morpho-syntaxe
  valeurs de propriétés, comme number, gender, mood, tense,
- syntaxe (valeurs de propriétés)
- morphologie
  valeurs de propriétés et classes de lettres
- phonétique, phonologie
  valeurs de propriétés et classes de sons (fricative, labiale, . . .)

Exemple : regexp 2 bandes

Reconnaissance d'un lemme à partir d'une forme et de règles morphologiques exprimées avec des transducteurs

```
:-finite_set(letters,[a,b,d,...,z]).
:-subset(voyelles,letters[a,e,i,o,u,y]).
:-subset(consonnes,letters[b,c,d,f,g,h,...,z]).
:-subset(doubles,consonnes[l,n,m]).

suffix(v1,
       ( c(X::doubles[]) : c(X),
         c(X) : true,                    %% même lettre doublée
         c(Y::voyelles[e]) : c(Y),  %% c(e) : c(Y) aussi OK
         true : c(consonnes[r])
       ),
       [ person: person[1,3],
         tense:  tense[present],
         mood:   mood[ind,subj]
       ]).

%% appe+lle => appe+ler    1|3.pres.ind|subj
```

Exemple (variante)

Exploitation plus agressive de l'unification immédiate !
partage des actions plutôt que partage des lettres

```
suffix(v1,
       ( X::c(doubles[]) : X,
         X : true,
         Y::c(voyelles[e]) : Y,
         true : c(consonnes[r])
       ),
       [ person: person[1,3],
         tense:  tense[present],
         mood:   mood[ind,subj]
       ]).

%% appe+lle => appe+ler    1|3.pres.ind|subj
```

Outline

Représenter des propriétés

Besoin naturel de représenter des ensembles de propriétés.

Plus élégant de nommer explicitement les propriétés, ce qui rend superflu :

- leur ordre d'apparition
- leur présence, quand non informative (information partielle)

$\Rightarrow$ même soucis dans la plupart des langages de programmation :

- `struct` en C
- `hash` en Perl (Python, Ruby, ...)
- listes de propriétés en Lisp

Notion de structures de traits en TAL

Notation avm

Notation matricielle des structures de traits, sous forme de Matrice Attributs Valeurs [AVM- *Attribute Value Matrix*]

$$\begin{bmatrix} \text{number} & \text{sg} \\ \text{person} & 2 \\ \text{tense} & \text{present} \\ \text{mood} & \text{indicative} \end{bmatrix} \equiv \begin{bmatrix} \text{person} & 2 \\ \text{mood} & \text{indicative} \\ \text{tense} & \text{present} \\ \text{number} & \text{sg} \end{bmatrix}$$

Récursion

La valeur d'une propriété peut être, récursivement, une structure de traits (AVM)

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Réentrance

Plusieurs propriétés peuvent partager une même valeur :

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Réentrance

Plusieurs propriétés peuvent partager une même valeur :

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

$$\equiv \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Réentrance (suite)

Quelques subtilités derrière la notion de réentrance :

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

$$\neq \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Réentrance & instanciation

La réentrance doit être préservée par instanciation :

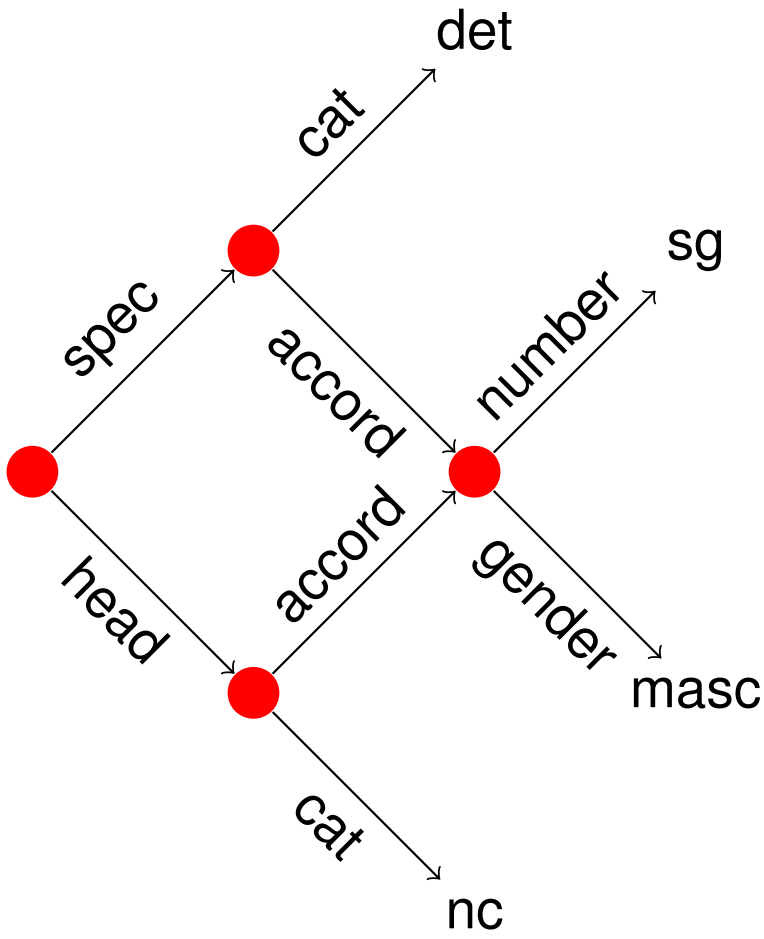
$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix} \rightsquigarrow \text{(inst.)} \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{fem} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix} \not\rightsquigarrow \text{(inst.)} \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \\ \text{gender} & \text{fem} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Notation graphe

Jusqu'à un certain point, les structures de traits sont représentables sous formes de graphes

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$



Note : intuitif de parler de chemin pour une séquence de traits dans le graphe et dans l'AVM

Unification

- Une structure de traits précise de l'information sur un objet, au travers des valeurs de traits

- L'unification de deux termes t_1 et t_2 accumule l'information $\rightsquigarrow t_1 \cup t_2$ (information plus spécifiée) – opération commutative et associative

- $\Rightarrow$ algorithme :
  - tout chemin p présent dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$
  - toute valeur pour un chemin p dans t_1 ou t_2 est présente dans $t_1 \sqcup t_2$
  - tout point de partage (réentrance) dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$

Unification

- Une structure de traits précise de l'information sur un objet, au travers des valeurs de traits

- L'unification de deux termes t_1 et t_2 accumule l'information $\rightsquigarrow t_1 \cup t_2$ (information plus spécifiée) – opération commutative et associative

- $\Rightarrow$ algorithme :
  - tout chemin p présent dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$
  - toute valeur pour un chemin p dans t_1 ou t_2 est présente dans $t_1 \sqcup t_2$
  - tout point de partage (réentrance) dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$

$$\begin{bmatrix} \text{Det} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Num sing} \end{bmatrix} \end{bmatrix} \\ \text{Nom} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Num sing} \end{bmatrix} \\ \text{Cat N} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{Nom} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Genre Masc} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Unification

- Une structure de traits précise de l'information sur un objet, au travers des valeurs de traits
- L'unification de deux termes t_1 et t_2 accumule l'information $\rightsquigarrow t_1 \cup t_2$ (information plus spécifiée) – opération commutative et associative
- $\Rightarrow$ algorithme :
  - tout chemin p présent dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$
  - toute valeur pour un chemin p dans t_1 ou t_2 est présente dans $t_1 \sqcup t_2$
  - tout point de partage (réentrance) dans t_1 ou t_2 est présent dans $t_1 \sqcup t_2$

$$\begin{bmatrix} \text{Det} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Num sing} \end{bmatrix} \end{bmatrix} \\ \text{Nom} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Num sing} \end{bmatrix} \\ \text{Cat N} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{Nom} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Genre Masc} \end{bmatrix} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{Det} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Num sing} \\ \text{Genre masc} \end{bmatrix} \end{bmatrix} \\ \text{Nom} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Num sing} \end{bmatrix} \\ \text{Cat N} \end{bmatrix} \end{bmatrix}$$

Unification (suite)

Cas avec réentrance :

$$\begin{bmatrix} \text{Det} \begin{bmatrix} \text{Accord } \boxed{1} \begin{bmatrix} \text{Num sing} \end{bmatrix} \end{bmatrix} \\ \text{Nom} \begin{bmatrix} \text{Accord } \boxed{1} \\ \text{Cat N} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{Det} \begin{bmatrix} \text{Accord} \begin{bmatrix} \text{Genre Masc} \end{bmatrix} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{Det} \begin{bmatrix} \text{Accord } \boxed{1} \begin{bmatrix} \text{Num sing} \\ \text{Genre masc} \end{bmatrix} \end{bmatrix} \\ \text{Nom} \begin{bmatrix} \text{Accord } \boxed{1} \\ \text{Cat N} \end{bmatrix} \end{bmatrix}$$

Subsomption

- une FS t_1 généralise (ou subsume) une FS t_2, noté $t_1 \sqsubseteq t_2$, ssi t_1 précise moins d'information que t_2
- (rappel) $\sqsubseteq$ est un pré-ordre partiel :
  $t \sqsubseteq t$; $t_1 \sqsubseteq t_2 \wedge t_2 \sqsubseteq t_3 \Rightarrow t_1 \sqsubseteq t_3$
- $\Rightarrow$ algorithme :
  - tout chemin dans t_1 est présent dans t_2
  - toute valeur pour un chemin p dans t_1 est présente pour p dans t_2
  - tout point de partage associé à un chemin p dans t_1 est présent dans t_2

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix} \sqsubseteq \begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{number} & \text{sg} \end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix} \text{gender} & \text{masc} \end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

Structures de traits ouvertes en Prolog

Les structures sont directement représentables comme des listes Prolog

$$\begin{bmatrix} \text{spec} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix}\text{number} & \text{sg}\end{bmatrix} \\ \text{cat} & \text{det} \end{bmatrix} \\ \text{head} & \begin{bmatrix} \text{accord} & \boxed{1}\begin{bmatrix}\text{gender} & \text{masc}\end{bmatrix} \\ \text{cat} & \text{nc} \end{bmatrix} \end{bmatrix}$$

```
?- FS = [spec: [accord: X::[number: sg],
                cat: det],
         head: [accord: X,
                cat: nc]
        ].
```

Efficacité

Divers problèmes d'efficacité avec la représentation ouverte en liste des FS :

- l'accès à la valeur d'un trait oblige à parcourir la list

```
fs_value(FS,F,Val) :- domain(F:Val,FS).
```

- l'unification de base Prolog n'est pas suffisante : il faut implanter une unification spécifique non efficace :

```
fs_unif(t1,t2) retourne FS
  t3 := []
  pour chaque f:v1 dans t1,
      si f:v2 dans t2,
      alors ajouter f:fs_unif(v1,v2) à t3
      sinon ajouter f:v1 à t3
  pour chaque  f:v2 dans t2,
      si f:v1 n'est pas dans t1,
      alors ajouter f:v2 à t3
  retourner t3
```

- $\Rightarrow$ nombreux parcours de listes :
  complexité en $O(n^2)$, n taille des structures
- Plus efficace si utilisation de listes triées de traits (ou arbres balancés)
- Sinon, codage en interne (en C) appelé à partir de Prolog

Manque de contrôle

Les FS ouvertes n'offrent pas de contrôle sur les traits introduits
$\Rightarrow$ facile de faire une erreur :

- sur le nom d'un trait : num vs number
- sur l'utilisation d'un trait dans un contexte inapproprié
  ex : par unification, ajout d'un trait tense dans la FS associée à un nom

Structures de traits fermées

Principe :

1. typer les structures de traits
2. associer à chaque type la liste (finie) des traits admissibles

Cela permet :

1. de contrôler les occurrences des traits (noms et contextes)
2. d'obtenir des représentations efficaces

Notation DyALog

Déclaration des types et traits admissibles :

```
:-features([nc,gn],[number,gender]).
:-features(v,[number,gender,person,tense,mood,voice,aux]).
:-features(gv,[number,gender,person, mood]).
```

Représentation :

```
s --> gn{ number => Num },
      gv{ number => Num, person => 3, mood => finite[]}.
gn{} --> GN::gn{}, rel(GN).
```

Représentation interne

La déclaration des traits permet :

- d'associer un rang à chaque trait pour un type donnée
- de représenter une FS fermée par un terme Prolog standard
- dans DyALog, la transformation est effectuée au moment de la lecture des FS, avec contrôle des traits :

```
DyALog> :-features(nc,[number,gender]).
DyALog> ?-X=nc{ num => sg }.
Line 2 of user_input:
        Syntax Error : feature num not allowed within feature
    structure nc{}
```

```
DyALog> ?-X=nc{ number => sg }, X =.. T.
        X = nc{number=> sg, gender=> B__5}
        T = [nc!$ft,sg,B__5]
```

La FS X=nc{ number => sg } représentée par nc!ft(sg,_)

- les valeurs non spécifiées sont remplis par des variables
- nc!ft est spécifique DyALog : foncteur ft dans le domaine de noms nc

Unification

Directement fournie par l'unification standard Prolog !

```
DyALog> ?-X=nc{ number => sg }, X = nc{ gender => masc }.
%% equiv: nc!ft(sg,_) = nc!ft(_,masc) = nc!ft(sg,masc)
        X = nc{number=> sg, gender=> masc}
```

pro et cons

Les structures de traits fermées sont

- \- moins flexibles que les FS ouvertes
- \+ plus sûres d'un point de vue contrôle
- \+ efficaces en utilisation,
- \+ pas de surcouche dans Prolog, (sauf dans le lecteur de termes)

Néanmoins, le contrôle n'est pas parfait !

- on peut se tromper dans la valeur d'un trait

```
?-X = v{ tense => indicative , number => plural }
%% au lieu de: v{ mood => ind , number => pl }
```

$\Rightarrow$ on souhaite pouvoir préciser le domaine de valeurs d'un trait, dans le contexte d'un type.

Outline

Hiérarchie de types

Les structures de traits typées [TFS – *Typed Feature Structures*] s'appuient sur une hiérarchie de types :

- un type τ peut avoir des sous-types $\tau_1, \ldots, \tau_n$
- un type τ peut être sous-type de plusieurs types parents (héritage multiple)
- un type τ peut introduire un trait f, en spécifiant le type $\rho_{f,\tau}$ le plus général approprié pour f (sous τ)
- un trait f est approprié pour un type τ si il est introduit par τ ou par un type ancêtre de τ

Formalisation développée par Bob Carpenter

- Implantation dans ALE en surcouche Prolog, Gerald Penn `http://www.cs.toronto.edu/~gpenn/ale.html`
- autre implantation dans LKB, Ann Copestake
- TFS surtout utilisées pour les HSPG
- Implantation dans DYALOG

Exemple

Fragment d'une hiérarchie pour une grammaire de génération (*Semantic-Head-Driven Generation*, Shieber et al, dans ALE)

```
bot sub [pred, list, sem, form, agr, sign].
 form sub [finite, nonfinite].
    finite sub [].
    nonfinite sub [].
 sign sub [sentence, verbal, np, adv, p]
       intro [sem:sem].
    sentence sub [].
    verbal sub [s, vp] intro [form:form].
      s sub [].
      vp sub [] intro [subcat:subcat_list].
    np sub [det, n]
       intro [agr:agr, arg:sem].
      det sub [] intro [np_sem:sem].
      n sub [].
    adv sub [] intro [varg:sem].
    p sub [].
 ...
```

Notation AVM pour les TFS

```
bot sub [pred, list, sem, form, agr, sign].
 sign sub [sentence, verbal, np, adv, p]
       intro [sem:sem].
    sentence sub [].
    verbal sub [s, vp] intro [form:form].
      s sub [].
      vp sub [] intro [subcat:subcat_list].
    np sub [det, n]
       intro [agr:agr, arg:sem].
      det sub [] intro [np_sem:sem].
      n sub [].
    adv sub [] intro [varg:sem].
    p sub [].
 ...
```

$$\text{np}\begin{bmatrix}\text{agr} & \text{sg1}\\ \text{arg} & \text{sem}\begin{bmatrix}\ldots\end{bmatrix}\\ \text{sem} & \text{sem}\begin{bmatrix}\ldots\end{bmatrix}\end{bmatrix}$$

Constantes et types

Version puriste des TFS :
les constantes sont aussi des types
maximaux (sans sous-types) et sans traits

```
bot sub [pred, list, sem, form, agr, sign].
  agr sub [sg1, sg2, sg3, pl1, pl2, pl3].
    sg1 sub []. sg2 sub []. sg3 sub [].
    pl1 sub []. pl2 sub []. pl3 sub [].
  pred sub [decl, imp, love, call_up, leave, see, ...].
    decl sub []. imp sub [].
    leave sub []. love sub []. call_up sub []. see sub [].
    ...
  ...
```

- déclarations assez lourdes, pour des ensembles larges de constantes
- quid des ensembles ouverts, comme les entiers ou les chaînes ?

Échappements

Dans DyALog, version non puriste des constantes
les constantes gérables par échappements pour revenir vers les types *builtins* Prolog :

```
bot sub [humain,name,age].
   humain intro [name: name, age: age].
   name escape symbol.
   age escape integer.
```

Types d'échappements : char, integer, symbol, et compound (tout terme)

Ne permet pas (encore) d'associer à un type un domaine fini de valeur

```
%% Dans un monde idéal
bot sub [v,mood].
   v intro [mood: mood, ...]
   mood escape finite_set(mood)
```

Termes squelettes

À chaque type τ peut être associé la TFS la plus générale possible compatible avec les contraintes de la hiérarchie.
$\Rightarrow$ notion de terme squelette $\mathrm{skel}(\tau)$

Plus précisément, si $f_1, \ldots f_n$ sont appropriés pour τ
avec les types $\rho_{f_1,\tau}, \ldots, \rho_{f_n,\tau}$, alors

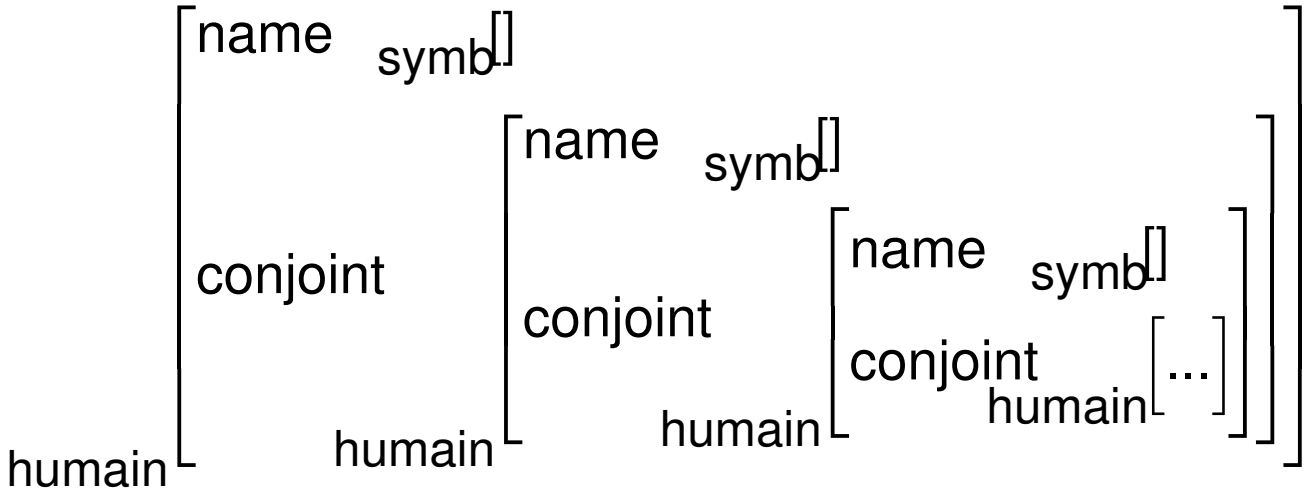
$$\mathrm{skel}(\tau) = \tau \begin{bmatrix} f_1 & \mathrm{skel}(\rho_{f_1,\tau}) \\ \ldots & \ldots \\ f_n & \mathrm{skel}(\rho_{f_n,\tau}) \end{bmatrix}$$

Type récursifs

Un type ne peut être directement récursif au travers d'un de ses traits, ie $\rho_{f,\tau} \neq \tau$

```
bot sub [humain, symbol]].
  humain intro [name: symbol, conjoint: humain].
  symbol escape symbol.
```

Motivation : conduit à un terme squelette infini



Dé-récursifier les types

Principe : introduire des types intermédiaires sous-spécifiés

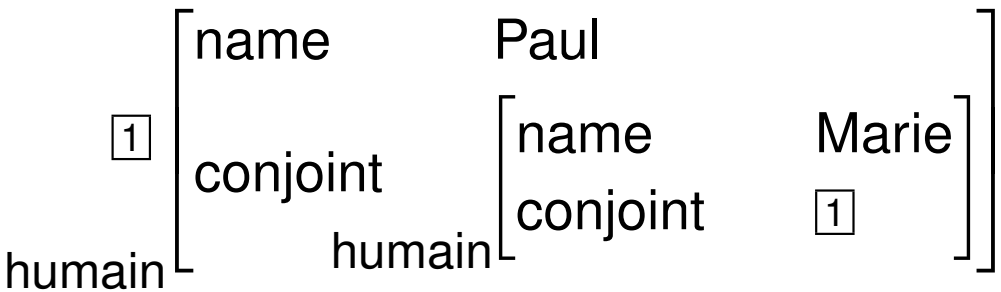
```
bot sub [humain, symbol]].
  humain sub concubin intro [name:symbol].
    concubin sub [] intro [conjoint: humain].
  symbol escape symbol.
```

Terme squelette pour concubin

$$\text{concubin}\left[\begin{array}{l}\text{name}\ \ \text{symb}[] \\ \text{conjoint}\ \ \text{humain}\left[\text{name}\ \ \text{symb}[]\right]\end{array}\right]$$

Termes récursifs

Une TFS peut être récursive et même cyclique sans que les types soient récursifs

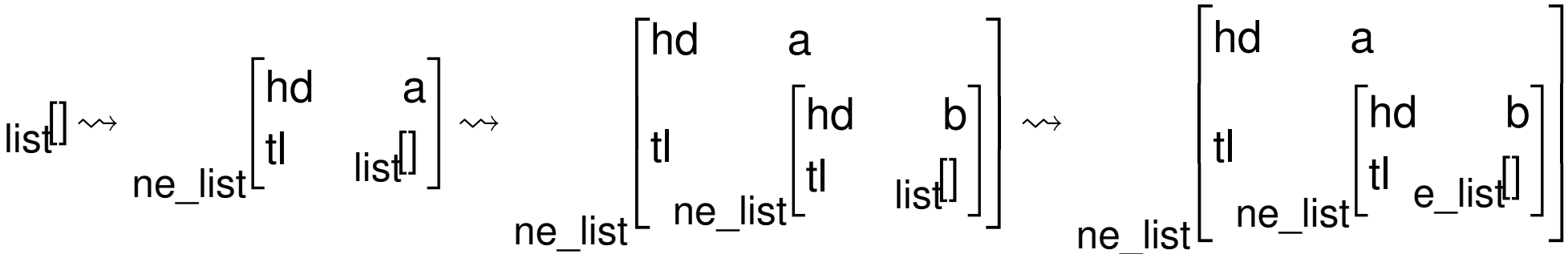


Note : la plupart des systèmes TFS refusent les termes cycliques, ils sont acceptés par DyALog

Listes

On peut définir la notion de liste avec une hiérarchie de type,
en utilisant un type dé-récursifié list

```
bot sub [list,symbol,...].
  list sub [e_list, ne_list].
    e_list sub [].
    ne_list sub []
            intro [hd:bot, tl:list].
  ...
```



Listes spécialisées

Souvent, on souhaite pouvoir spécifier une liste d'éléments d'un certain type :

- non directement possible d'utiliser des listes paramétriques X list
- mais possible en sous-typant

```
list sub [e_list, ne_list, arg_list, subcat_list].
  e_list sub [].
  ne_list sub [arg_ne_list, subcat_ne_list]
          intro [hd:bot, tl:list].
  arg_list sub [e_list, arg_ne_list].
    arg_ne_list sub [] intro [hd:sem, tl:arg_list].
  subcat_list sub [e_list, subcat_ne_list].
    subcat_ne_list sub [] intro [hd:sign, tl:subcat_list].
```

Listes et AVM

Les listes sont une notion suffisamment importante pour bénéficier d'une notation spéciale dans AVM :

```
verbal sub [s, vp] intro [form:form].
  s sub [].
  vp sub [] intro [subcat:subcat_list].
```

$$\mathrm{vp}\left[\begin{array}{ll} \mathrm{sem} & \mathrm{sem}[] \\ \mathrm{form} & \mathrm{form}[] \\ \mathrm{subcat} & \langle [\mathrm{np}],[\mathrm{np}],[\mathrm{pp}] \rangle \end{array}\right]$$

Contraintes sur les hiérarchies (suite 1)

- un type τ ne doit pas avoir de trait approprié f tel que $\rho_{f,\tau} = \tau$ (pas de type récursif)

Contraintes sur les hiérarchies (suite 2)

- un type τ_1 peut ré-introduire un trait *f* avec un type ρ_{f,τ_1}
  (déjà introduit par un ancêtre avec le type $\rho_{f,\tau}$)
  mais alors ρ_{f,τ_1} doit être un sous-type de $\rho_{f,\tau}$
  (instanciation des types appropriés)

```
ne_list sub [arg_ne_list, subcat_ne_list]
        intro [hd:bot, tl:list].
subcat_list sub [e_list, subcat_ne_list].
  subcat_ne_list sub [] intro [hd:sign, tl:subcat_list].
```

Contraintes sur les hiérarchies (suite 2)

- si τ_1 et τ_2 introduisent le trait *f*,
  alors un type ancêtre τ de τ_1 et τ_2 doit introduire *f*
  $\Rightarrow$ existence d'un type plus général σ_f introducteur de *f* Cas de hd et tl

```
list sub [e_list, ne_list, arg_list, subcat_list].
  e_list sub [].
  ne_list sub [arg_ne_list, subcat_ne_list]
          intro [hd:bot, tl:list].
  arg_list sub [e_list, arg_ne_list].
    arg_ne_list sub [] intro [hd:sem, tl:arg_list].
  subcat_list sub [e_list, subcat_ne_list].
    subcat_ne_list sub [] intro [hd:sign, tl:subcat_list].
```

Unification

L'unification entre TFS est :

- similaire à l'unification entre FS fermée
- mais doit prendre en compte l'instantiation vers des sous-types
- $\Rightarrow$ étend strictement l'unification Prolog

Algorithme :

```
tfs_unif(t1 type τ1, t2 type τ2) retourne TFS
   τ3 := mgst(τ1,τ2) ou fail
   t3 := [{τ3}]
   pour tout f dans t1 avec v1 et t2 avec v2,
        t3 += f: v1 ⊓ v2 ⊓ skel(ρf,τ3)
   pour tout f dans t1/t2 avec v1
        t3 += f: v1 ⊓ skel(ρf,τ3)
   pour tout f dans t2/t1 avec v2
        t3 += f: v2 ⊓ skel(ρf,τ3)
   pour tout f approprié pour τ3, non dans t1 ou t2
        t3 += f: skel(ρf,τ3)
   retourne t3
```

Unification : implantation

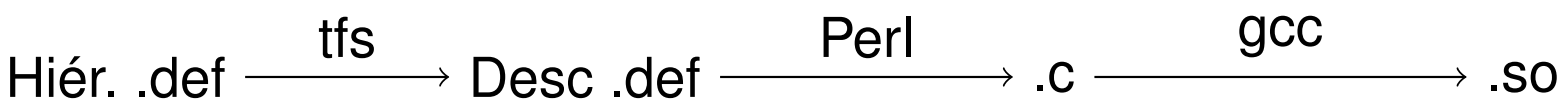
L'unification entre TFS étend strictement l'unification entre termes Prolog

On peut implanter l'algorithme d'unification de manière générique en surchouche de Prolog

Mais l'information de la hiérarchie de type permet en fait de définir hors-ligne une opération d'unification spécialisée par paire de type τ_1 et τ_2.

Compilation : tfs2lib

DyALog compile les hiérarchies de types en bibiothèque C dynamique en utilisant TFS2LIB



Exploitation :

```
dyacc -tfs list tfs_append.pl -o tfs_append
```

Description expansée

TFS produit une description expansée d'une hiérarchie de types

```
TYPES = p adv n det np vp s verbal ...  bot
SUBTYPES np = det n
SUBTYPES verbal = s vp
SUBTYPES sign = s vp det n sentence verbal np adv p
SUBTYPES agr = sg1 sg2 sg3 pl1 pl2 pl3
SUBTYPES form = finite nonfinite
SUBTYPES list = arg_ne_list subcat_ne_list e_list ne_list
    arg_list subcat_list
...
FEATURES = varg np_sem arg agr subcat form sem args pred tl hd
INTRO varg = adv
INTRO np_sem = det
INTRO tl = ne_list
INTRO hd = ne_list
..
```

Description expansée (skel)

```
SKEL p = sem:sem
SKEL adv = varg:sem sem:sem
SKEL n = arg:sem agr:agr sem:sem
SKEL det = np_sem:sem arg:sem agr:agr sem:sem
SKEL np = arg:sem agr:agr sem:sem
SKEL vp = subcat:subcat_list form:form sem:sem
SKEL sem = args:arg_list pred:pred
SKEL subcat_ne_list = tl:subcat_list hd:sign
SKEL subcat_list =
SKEL arg_ne_list = tl:arg_list hd:sem
SKEL arg_list =
SKEL ne_list = tl:list hd:bot
...
```

Description expansée (unif)

```
UNIF bot list = list
UNIF bot e_list = e_list
UNIF sign vp = vp
+       UNIF sem LEFT 1 RIGHT 3 DIRECT 3
+       INHERIT form RIGHT 2 DIRECT 2
+       INHERIT subcat RIGHT 1 DIRECT 1
...
```

Note :

- les fonctions exploitent le fait qu'un rang peut être associé à chaque trait pour un type donné.
- des fonctions de subsumption sont aussi engendrées

Représentation interne (DyALog)

- Même représentation que pour les FS fermées, pour les types maximaux
- Passage en deref-termes avec un argument supplémentaire pour les termes non maximaux
  permet l'instantiation de types

Notation en chemin

Les informations de la hiérarchie permettent des inférences de termes à partir de chemins de traits.

```
append(e_list{}, Y, Y).

append( A::tl=>X,Y::list{}, B::tl=>Z) :-
        A .> hd .= B.> hd,
        append(X,Y,Z).
```