Tutorial : very preliminary version 0.1 7/5/2002

Couples, noyau et structure de données de DefOnto

Introduction

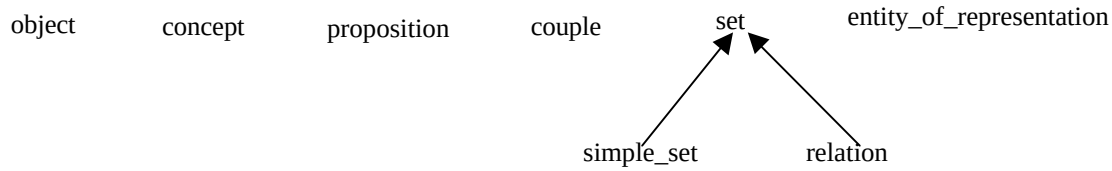
DefOnto est un sous ensemble de DefEtoile, dédié à la mise en œuvre d'ontologies.
Une base de connaissances (BC) met en œuvre des couples d'entités de représentation , d'objets, de concepts, de propositions. Ces couples sont étiquetés par un marqueur de relation.
Un couple est déterminé par les deux entités qu'il relie, prédécesseur et successeur, et la relation qui étiquette ce couple. Par la suite une instance de couple sera noté:
couple ::= [#identificateur | (couple_prédecesseur)] Q1Q2#[identificateur | id_couple] #[identificateur | (couple_successeur) | terminal | typeTerminal]
couple_prédecesseur ::= [#identificateur Q1Q2#identificateur #[identificateur | terminal | typeTerminal]
couple_successeur ::= [#identificateur Q1Q2#identificateur #[identificateur | terminal | typeTerminal]
L'objectif de ce texte est de développer une classification susceptible d'une implantation dans une structure de données classique, telle un graphe orienté de réseau.

Composition du noyau

Il est constitué de primitives non définies et de primitives définies dont la sémantique est prédéfinie pour le calcul des inférences:

- Primitives non définies de premier niveau : entity_of_representation, couple, object, concept, proposition, set;
- Primitives non définies de deuxième niveau : simple_set et relation sont subsumées set;
- Primitives de services inférentiels non redéfinissables.

Graphe des primitives non définies



Sont rattachées à ce graphe l'ensemble des objets du noyau et de la base de connaissances. L'ingénieur de la connaissances peut ajouter des bibliothèques génériques à tous les domaines ou à certains domaines, telle la relation #has_for_main_term.

Concept, couple et proposition

Les catégories de DefOnto, concept, couple et proposition sont à analyser préalablement à toute tentative de construction de structures de données.
Ainsi les concepts individuels et génériques, les relations du langage DefOnto sont des concepts au sens général du terme. Les couples sont en principe des propositions (concepts assertifs). Les relations sont des ensembles de couples. On ne trouvera donc pas dans le langage le type de définition DefOnto.
Toute proposition est un couple, mais l'inverse n'est pas toujours vérifié. Tous les couples qui ont pour prédécesseur un couple, complètent la proposition formée par couple, mais ne sont pas eux-mêmes des couples.

Exemple:
#french AE#drinks #wine est un couple qui forme une proposition.
(#french AE#drinks #wine) #quantity "a lot" est un couple qui complète la proposition précédente, mais ce couple ne constitue pas une proposition.
(#french AE#drinks #wine) #proposition_belongs_to #stereotype) est un couple et non pas une proposition.

Contenu du noyau

http://www.laria.u-picardie.fr/EQUIPES/ic/DefOnto/Kernel.in

Classification des couples de DefOnto

Dans la syntaxe actuelle certains couples de la BC sont décrits littéralement. D'autres sont simplement une traduction d'un sucre syntaxique ou de l'analyse d'une CNS. D'autres sont considérés comme déduits selon le type de définition pour maintenir la cohérence de la BC. D'autres enfin sont déduits des couples, toujours pour maintenir la cohérence de la BC.

Exemple :
```
DefGenConcept #french
=[#person]->(MI#has_for_nationality)->"french"
PropertiesObjects
->(AE#eats)-
->[#snail]
->(#proposition_belongs_to)->[#stereotype])
```

Les couples décrits littéralement:
- #french AE#eats #snail
- (#french AE#eats #snail) #proposition_belongs-to #stereotype

Les couples traduits (IsA, CNS):
- #french #is_included_in #person (on considère qu'il y eu transformation selon le contexte, donc un calcul)
- #french AI#has_for_nationality "french" (partie CN de la CNS)

Les couples déduits des définitions de concept:
- #french #concept_belongs_to #concept
- #french #set_belongs_to #simple_set
- #french #entity_belongs_to #entity_of_representation

Les couples déduits des couples
- (#french AI#has_for_nationality "french") #entity_belongs_to #entity_of_representation
- (#french AE#has_for_nationality "french") #proposition_belongs_to #proposition
- (#french #is_included_in #person) #entity_belongs_ito #entity_of_representation
- (#french #is_included_in #person) #proposition_belongs_to #proposition
- (#french AE#eats #snail) #entity_belongs_to #entity_of_representation

Couples traduits

La traduction d'un couple s'entend comme une interprétation sémantique de la syntaxe du langage DefOnto. Les couples sont crées par:
1. les transformations du sucre syntaxique IsA
2. transformation par interprétation d'une CNS

Traduction du sucre syntaxique "IsA"

Ce sucre syntaxique se traduit par les relations #is_included_in ou #object_belongs-to selon que l'on a affaire à un concept générique, à une relation ou à un concept individuel. Le tableau suivant illustre les deux cas qui peuvent se présenté. La relation d'appartenance s'applique aux concepts individuels uniquement.

Def	IsA
Gen | Relation	is_included_in
IndConcept	object_belongs_to

Exemple :
```
(DefGenConcept #drink
IsA [#food])
```

le couple traduit :
#drink #is_included_in #food

```
(DefIndConcept #jules
IsA [#man],[#child])
```

Les couples traduits sont :
#jules #object_belongs_to #man

#jules #object_belongs_to #child

Interprétation d'une CNS

La traduction d'une condition nécessaire et suffisante qui ne s'applique qu'aux concepts génériques engendre:

- un couple décrit
- un couple traduit
- une règle

Exemple:

```
DefGenConcept #french
=[#person]->(MI#has_for_nationality)->"french")
```

Le couple décrit :
#french AI#has_for_nationality "french"
Ce couple correspond à la condition nécessaire (CN). La condition suffisante (CS) est une règle de la forme:
si [*person *x] et [has_for_nationality #x "french"] alors [#french *x].

Le couple traduit:
#french #is_included_in #person.

Dans l'exemple suivant la relation "object_belongs_to devient dans le réseau sémantique "is_included_in".
(DefGenConcept #father
=[#parents]->(MI#object_belongs_to)->[#man]

Couples déduits des définitions de concepts

Selon les types de définitions, des couples supplémentaires sont déduits selon certaines correspondance avec les types de définitions de concept. Le tableau suivant résume les modalités de déductions de ces couples supplémentaires.

TypeDef	#entity_belongs_to	#concept_belongs_to	#set_belongs_to	#proposition_belongs_to	sur types
MetaGenConcept	#entity_of_representation	#concept	#simple_set		concept
GenProposition	#entity_of_representation	#concept	#simple_set		proposition
GenEntity	#entity_of_representation	#concept	#simple_set		entity
GenConcept	#entity_of_representation	#concept	#simple_set		object
Relation	#entity_of_representation	#concept	#relation	#proposition	couple
IndConcept	#entity_of_representation	#concept			

Exemple :

```
(DefGenConcept #person
IsA #animate_being)
```

Les couples décrits :
#person #is_included_in #animate_being

Les couples déduits :
#person #concept_belongs_to #concept
#person #set_belongs_to #set
#person #entity_belongs_to #entity_of_representation

Les sur types traduisent les relations d'appartenance suivantes:
if(type.equals("DMGC")) group = "concept_belongs_to";
if(type.equals("DGP")) group = "proposition_belongs_to";
if(type.equals("DGE")) group = "entity_belongs_to";
if(type.equals("DGC")) group = "object_belongs_to";
if(type.equals("DR")) group = "couple_belongs_to";
if(type.equals("DIC")) group = "null";

Couples déduits de couples

Tout couple constitue une entité de représentation et une proposition. Les couples déduits de cette règle sont ajoutés à la BC.

Exemple :

```
DefGenConcept #french
=[#person]->(MI#has_for_nationality)->"french"
PropertiesObjects
->(AE#eats)-
->[#snail]
->(#proposition_belongs_to)->[#stereotype])
```

(#french #is_included_in #person) #entity_belongs_ito #entity_of_representation
(#french #is_included_in #person) #proposition_belongs_to #proposition
(#french AE#has_for_nationality "french") #entity_belongs_to #entity_of_representation
(#french AE#has_for_nationality "french") #proposition_belongs_to #proposition
(#french AE#eats #snail) #entity_belongs-to #entity_of_representation
(#french AE#eats #snail) #proposition_belongs-to #proposition
Dans cet exemple #proposition_belongs_to se substitue à la déduction normale en rattachant le couple #french AE#eats #snail à la proposition #stereotype.
(#french AE#eats #snail) #proposition_belongs-to #stereotype. C'est une redéfinition de la proposition (#french AE#eats #snail). Le couple déduit (#french AE#eats #snail) #proposition_belongs-to #proposition se trouve "surchargé" par (#french AE#eats #snail) #proposition_belongs-to #stereotype.

Cohérence interne de DefOnto

La cohérence de la description d'une BC dans le langage DefOnto est assurée par le fait que toute définition est liée directement ou par héritage à une primitive non définie. La classification des propriétés en fonction de chaque définition obéit à la syntaxe du langage. En matière de relation, lorsque celle-ci n'est pas une subsomption d'une autre relation, la nature du prédécesseur détermine le lien avec le noyau.

Définitions et propriétés

Le langage DefOnto associe à chaque type de définition un ensemble de propriétés qui constituent l'expression d'autant de conditions nécessaires.
La grammaire de DefOnto (permet de résumer les règles d'association des propriétés pour chaque type de définition dans le tableau suivant.

Properties	DMCG	DGP	DGE	DGC	DR	DIC
Concept	X	X	X	X	X	X
Concepts	X					
Couples					X	
Entities			X			
Entity	X	X	X	X	X	X
Object						X
Objects				X		
Propositions		X				
Relation					X	
Set	X	X	X	X		

La lecture de ce tableau est immédiate. Un concept individuel, par exemple, représente à la fois un objet, un concept et une entité de représentation. Des relations peuvent donc considérer ce concept individuel comme un concept ou un objet.
Un métaconcept générique est un concept, une entité, un groupe et un ensemble de concepts.

Relations

Toute définition de relation qui n'est pas subsumée par une autre relation hors du noyau doit être reliée au noyau. Le tableau suivant indique en fonction de la nature du prédécesseur du couple et du premier quantificateur, le successeur du couple formé par cette relation.

has_for_domain	Quantificateur Q1	couple_which_predecessor_is_X
concept	I	concept
	A | E	
Etiquette de couple	A | E	couple
entity	I	entity
	A | E	
object	I	object
	A | E	
proposition	A | E	proposition
set	I	set

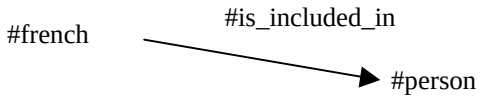
A = all, I = Individual, E = Exist

Exemple:
```
(DefRelation #has_for_main_term
IsA [#couple_which_predecessor_is_concept]
PropertiesRelation
-> (#has_for_domain) -> [#concept])

(DefRelation #has_for_nationality
IsA [#couple_which_predecessor_is_object]
PropertiesRelation
->(#has_for_domain)-> [#person]
->(#has_for_range)->Tstring)

(DefRelation #since
IsA [#couple_which_predecessor_is_couple]
PropertiesRelation
->(#has_for_domain)->[#has_for_nationality]
->(#has_for_range)->Tstring)
```

Structure de données

C'est un réseau dont les composants élémentaires sont des couples de sommets reliés par des arcs orientés. Il existe un dictionnaire des identificateurs qui regroupe l'ensemble des entités de représentation, un dictionnaire des arcs qui rassemble les identificateurs de relation.
Chaque arc peut être référencé comme un sommet.
Chaque objet du réseau, arc et sommet, est pourvu d'un dictionnaire qui contient l'ensemble des propriétés de l'objet.
L'exécution des filtres est fondée sur des primitives classiques de parcours de graphe.

Chaque couple se représente à l'aide de deux sommets relié par un arc. La construction des couples est immédiate.



Par contre la traduction des CNS nécessite quelques explications.
Construction d'une CNS
Rappelons qu'une CNS se traduit à l'aide de deux couples et une règle:

Exemple:
```
DefGenConcept #french
=[#person]->(MI#has_for_nationality)->"french")
```

Le couple décrit :
#french AI#has_for_nationality "french"
Ce couple correspond à la condition nécessaire (CN). La condition suffisante (CS) est une règle de la forme: si [#person *x] et [has_for_nationality #x "french"] alors [#french *x].

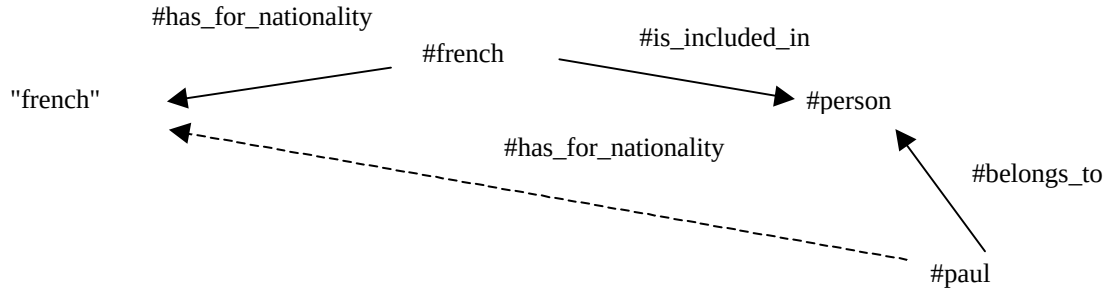
Le couple traduit:
#french #is_included_in #person

Soit la définition suivante :
(DefIndConcept #paul
IsA [#person]
PropertiesObject
->(#has_for_nationality)->"french")

Le couple décrit:
#paul #has_for_nationality "french"

Le couple traduit:
#paul #belongs_to #person

Représentation :



Interprétation graphique de la règle si [#person *x] et [has_for_nationality #x "french"] alors [#french *x]
paul est une personne
paul a pour nationalité la nationalité française
paul est un français car c'est une personne de nationalité française

Résolveur de CS

Soit la définition suivante:

```
(DefGenConcept #greedy
=[#person]->(ME#consumes)-
->[#food]
->(#in_quantity)->"much " ,
PropertiesConcept
->(#has_for_main_term)->"greedy")
```

Résultat de la compilation:

```
********************** Def DGC greedy ****************************
### greedy > is_included_in > person
# greedy > consumes > food
# CP greedy consumes food > in_quantity > ~much
```

Résolveur CS:
La règle :
si [#person *x] et [consumes *c *x *y] et [#food *y] et [#in_quantity *c much]
alors [#greedy *x]

Dans les filtres concernés (voir section Computing), les variables sont instanciées.
Les couples précédents ont le numéro de CNS 17 (étiquette de l'arc).

Conditions :

```
greedy > is_included_in > person
```

[#person *x]
En entrée *x:
louis
En sortie:
#person #louis VRAI

`greedy > consumes > food`
peut-déduire-que[#consumes #instance #instance] pour chaque entrée filtre 12
En entrée *x:
consumes louis
En sortie on construit la collection si VRAI:
consumes louis potatoes
consumes louis wine
consumes louis apple
consumes louis ~tobacco

peut-déduire-que[#food #instance] pour chaque entrée filtre 4
En entrée *y:
potatoes
wine
apple
~tobacco
En sortie on construit la collection si VRAI::
potatoes
wine
apple

Sélection sur la première collection à partir de la deuxième.

`greedy consumes food > in_quantity > ~much`
peut-déduire-que[#in_quantity #instance much] pour chaque entrée filtre 12
En entrée *c:
consumes louis potatoes
consumes louis wine
consumes louis apple
En sortie:
Si le filtre renvoie une collection non vide alors conclusion louis greedy VRAI
Sinon louis greedy FAUX

Inference services

Data processing before computing

global-adding-belonging-links

This procedure aims to complete the description of individual entities before computing. it uses the signatures of the relations and the properties of the individual entities to infer new membership links between these entities and types. Five membership relations are used :

- `#object-belongs-to`
- `#concept-belongs-to`
- `#proposition-belongs-to`
- `#entity-belongs-to`
- `#set-belongs-to`

Before adding a new membership link, a verification is made that this link can't be infer from an existing membership. For example if we know than Mary is a woman, it is useless to add she is a person.

individual-concept-adding-belonging-links

This procedure aims to complete the description of an individual concept before computing. It's goal is the same as the previous procedure, simply the process is limited to examine the description of an individual concept (the

contents of a complete definition of concept by means of the construction `DefIndConcept`). The procedure uses the properties linked to the individual concept and the signatures of the corresponding relations to infer new links of membership between the individual concept and types.
Three membership relations are used :

- `#object-belongs-to`
- `#concept-belongs-to`
- `#entity-belongs-to`

Before adding a new membership link, , a verification is made that this link can't be infer from an existing membership, as for the previous procedure.
Note than properties of individual concept may match to propositions. Then, a look at properties associated to the proposition indicates if it is possible to determine the membership of the property to a more specific class of propositions.

couples-inheriting

This procedure carries out the inheritance of couples owning the `AI` quantifier along the taxonomy of generic concepts respectively which have as root :

- `#object`
- `#concept`
- `#proposition`
- `#entity_of_representation`
- `#set`

As well, the inheritance is made at level of instances of those generic concepts, the couples with `AI` quantifier becomes couples with quantifier `II`.

Computing

[#type *x]

The return value of this filter is the direct extension of the concept `#type`. According to the kind of the concept `#type`, the instances are linked to the concept by one among the next relations:

```
#object-belongs-to
#concept-belongs-to
#proposition-belongs-to
#entity-belongs-to
#set-belongs-to
```

(can-infer-from[#type *x])

The boolean return value of this filter is the computing extension of the concept `#type`, which considers all possible inferences in the whole lattice of generic concept.

[#type #instance]

The boolean return value of this filter indicates if the entity `#instance` has for direct type the generic concept `#type`, according the kind of the concept `#type` among the next relations:

```
#object-belongs-to
#concept-belongs-to
#proposition-belongs-to
#entity-belongs-to
#set-belongs-to
```

(can-infer-from[#type #instance])

The boolean return value says if the entity `#instance` has for computed type (taking into account the whole lattice of generic concepts) the generic concept `#type`.
This evaluation leads to search admissible chains composed of links of the relation `#is_included_in`, between the types of the entity `#instance` and `#type`.

A Chain of links `#is_included_in` between concept1 and concept2 is an ordering list of generic concepts(cpt_1, cpt_2, …, cpt_n) like cpt_1=concept1;cpt_n=concept2 and $\forall i \in \{1,\ldots n\}$ cpt_i `#is_included_in` cpt_{i+1} where cpti+1 `#is_included_in` cpt_i
this chain is *admissible* if
it is elementary, that means a same concept can appear only once : $\neg\exists i, j \in \{1, \ldots$ nlike $i \neq j$ et $cpt_i = cpt_j$.
$\forall i \in \{1, \ldots n\}$, if cpt_{i+1} `#is_included_in` cpt_i then necessarily cpt_{i+1} is a defined generic concept (there is a CNS associated to cpt_{i+1}) compared to cpt_i.
if $\exists i \in \{1, \ldots n\}$ like cpt_{i-1} `#is_included_in` cpti and cpt_{i+1} `#is_included_`in cpt_i then necessarily cpt_{i-1} and cpt_{i+1} are not inconsistent. (ex : `#man` et `#woman` are inconsistent under the concept `#person`).

[#relation *x]

It is the direct extension of the relation `#relation`.

(can-infer-from[#relation *x])

The return value gives the computing extension of the relation `#relation`, taking account into the whole tree of the relations.

[#relation #instance *y]

The value returned is the direct set of successors of the entity `#instance` for the relation `#relation`.

(can-infer-from[#relation #instance *y])

Computing extension of successors of the entity `#instance`, taking account into the tree of the relations.

[#relation *x #instance]

Direct set of predecessors of the entity `#instance` for the relation `#relation`.

(can-infer-from[#relation *x #instance])

Computing extension of predecessors of the entity #instance for the relation `#relation`, considering the tree of the relations.

[#relation #instance1 #instance2]

Return boolean value, means that a couple with predecessor `#instance1` and for successor `#instance2` is a couple of the relation #relation.

(can-infer-from[#relation #instance1 #instance2])

Same as previous, but taking account into tree of the relations.

System configuration

File system

```
c:\Mesdoc~1\DefOnto\DefOntoWork

DefOntoWork --> sources
                        --> DefOntoAntlr
                        --> DefOntoInfer

            --> grammars
            --> classes
                        --> DefOntoAntlr
                        --> DefOntoInfer
            --> defFiles
            --> workFiles
```

Packages

```
java and antlr
c:\jdk1.3.1_01
c:\antlr\antlr-2.7.1

GFC and JSX
DefOntoWork\GFC         graphs
DefOntoWork\JSX         serialization
set CLASSPATH=c:\mesdoc~1\defonto\defontoWork\GFC\java\classes;
      c:\mesdoc~1\defonto\defontoWork\JSX;%CLASSPATH%
```

Parser, lexer

To build lexer, parser, AST files (when sources will be available)
antlr.bat in sources\DefOntoAntlr

```
c:\jdk1.3.1_01\bin\java -classpath .;
      c:\antlr\antlr-2.7.1;
      c:\Mesdoc~1\DefOnto\DefOntoWork\GFC\java\classes;
      antlr.Tool -o sources\DefOntoAntlr grammars\%1.g > compilation
antlr DefOntoLexis_1-2
antlr DefOntoGrammar_1-26
antlr DefOntoTree_1-30  results in DefOntoWork/workFiles
```

To build HTML grammar
antlrHTML.bat in sources\DefOntoAntlr

```
c:\jdk1.3.1_01\bin\java -classpath .;
      c:\antlr\antlr-2.7.1 antlr.Tool -o sources\DefOntoAntlr -html
      grammars\%1.g
antlr DefOntoGrammar_1-26
```

To compil antlr files
compil.bat in sources\DefOntoAntlr files (when sources will be available)

```
c:\jdk1.3.1_01\bin\javac -g -deprecation -classpath .;c:\antlr\antlr-2.7.1
-d ..\..\classes\DefOntoAntlr *.java
```

To parse Tree DefOnto files
run.bat in classes\DefOntoAntlr

```
c:\jdk1.3.1_01\bin\java -classpath .;
      c:\antlr\antlr-2.7.1 Main ..\..\DefFiles\%1.in >
      ..\..\DefFiles\person.err
In classes\DefOntoAntlr
run person
```

Inferences

To compile inferences files (when sources will be available)

```
c:\jdk1.3.1_01\bin\javac -g -deprecation -classpath .;c:\antlr\antlr-2.7.1
-d ..\classes\DefOntoInfer *.java
```

To run filters

```
runOntoInfer.bat in classes\DefOntoInfer
c:\jdk1.3.1_01\bin\java -classpath
.;c:\mesdoc~1\defonto\defontowork\GFC\java\classes OntoInfer
```