INF 424

TP 2 de logique

Yannis Haralambous (Télécom Bretagne)

Nous allons utiliser le package Python NLTK. Pour l'installer sans être root, on écrira on écrira

```
pip install nltk --user
```

Attention, dans ce TP on utilise la version 3 de NLTK, si vous avez la version 2, il faudra écrire

```
pip install --upgrade nltk --user
```

1 Preuve logique

La classe `nltk.LogicParser` fournit un *parseur logique*. La méthode `parse` permet de créer des objets représentant des formules logiques. La classe `nltk.Prover9` fournit un démonstrateur automatique de théorème. En lui fournissant les objets-formule il peut nous trouver une preuve et la détailler.

(Pour installer le démonstrateur, récupérer
`http://www.cs.unm.edu/~mccune/prover9/download/LADR-2009-11A.tar.gz`,
le décompresser, faire `make all`. Si le chemin d'accès de votre répertoire `LADR-2009-11A` est `/users/*****/LADR-2009-11A`, alors on définira une variable d'environnement `PROVER9` qui prend la valeur `/users/*****/LADR-2009-11A/bin`.

Voici la syntaxe dans laquelle il faut écrire les formules :

- `&` conjonction
- `|` disjonction
- `-` négation
- `->` implication
- `<->` double implication
- `all` quantificateur universel
- `exists` quantificateur existentiel

Prenons l'exemple classique : *Tout humain est mortel. Socrate est un humain. Montrer que Socrate est un mortel*, autrement dit :

- $\forall x\, \text{humain}(x) \to \text{mortel}(x)$
- $\text{humain}(socrate)$
- Mq $\text{mortel}(socrate)$.

Il s'écrira

```
import os
os.environ['PROVER9']='/users/*****/LADR-2009-11A/bin'
import nltk
lp = nltk.sem.logic.LogicParser()
A = lp.parse('all x. (humain(x) -> mortel(x))')
B = lp.parse('humain(socrate)')
```

```
C = lp.parse('mortel(socrate)')
prover = nltk.Prover9()
print prover.prove(goal=C,assumptions=[A,B],verbose=True)
```

et va donner, entre autres :

```
1 (all x (humain(x) -> mortel(x))) # label(non_clause).  [assumption].
2 mortel(socrate) # label(non_clause) # label(goal).  [goal].
3 humain(socrate).  [assumption].
4 -humain(x) | mortel(x).  [clausify(1)].
5 mortel(socrate).  [resolve(3,a,4,a)].
6 -mortel(socrate).  [deny(2)].
7 $F.  [resolve(5,a,6,a)].
============================== end of search =========================
THEOREM PROVED
```

Ici «assumption» signifie «hypothèse de départ», «goal» = «requête», «clausify» = ré-écriture en CNF avec éventuellement une skolémisation, «resolve(3,a,4,a)» = effectuer une résolution qui élimine le premier (d'où le «a») terme de la formule 3 et le premier terme de la formule 4, «deny» = négation de la requête. Le `$F` de la fin signifie que $\mathrm{KB} \wedge \alpha = \emptyset$.

1.1 Exercice

Appliquer cette techno au problème policier du cours :

1. *Tous ceux qui aiment tous les animaux sont aimés par qqun.*
2. *Quiconque tue un animal n'est aimé par personne.*
3. *Jack aime tous les animaux.*
4. *C'est soit Jack soit Curiosité qui a tué le chat appelé Luna.*
5. *Montrer que c'est Curiosité qui a tué le chat.*

2 Traduction de la langue naturelle en formalisme logique

Reprenons la modélisation de la langue naturelle amorcée dans le TP 1. Nous allons dans la suite pousser plus loin ces méthodes en nous basant sur le principe suivant, appelé «principe de compositionnalité» : *La sémantique d'une phrase s'obtient à partir des sémantiques de ses parties et de la manière dont elles ont été composées (= la syntaxe de la phrase).*

Notre but sera de traduire en formalisme logique *tous* les mots d'une phrase. Par le principe de compositionnalité, ces traductions vont se combiner entre elles pour produire la traduction de la phrase toute entière.

2.1 Théorie des types

Astuce : pour savoir comment formaliser les différents mots d'une phrase, voyons ce qu'ils deviennent lorsqu'on interprète la formule (dans le sens d'«interprétation de formule logique» comme on l'a vu en cours).

Un exemple : dans la phrase *Gérard aime Alice*, *Gérard* est naturellement interprété par un élément «Gérard» du domaine D, et de même pour *Alice*. L'interprétation du prédicat binaire *aime* peut être considérée comme une application $D^2 \to \{\text{vrai}, \text{faux}\}$ (elle envoie la paire d'entités (Gérard, Alice) vers la valeur vrai si Gérard aime Alice et vers faux sinon).

Autre exemple : dans la phrase *Gérard aime Alice et Paul déteste Virginie*, la particule de coordination «*et*» va combiner deux phrases pour en produire une nouvelle, son interprétation sera donc une application $\{\text{vrai}, \text{faux}\} \times \{\text{vrai}, \text{faux}\} \to \{\text{vrai}, \text{faux}\}$ dont la table de valeurs correspond à celle du connecter $\wedge$.

La situation se complique encore plus dans le cas des adverbes : dans *Gérard aime beaucoup Alice*, l'adverbe *beaucoup* agit sur le verbe, donc on peut considérer qu'il transforme une application $D^2 \to \{\text{vrai}, \text{faux}\}$ en une autre application $D^2 \to \{\text{vrai}, \text{faux}\}$.

Et que dire alors des modificateurs d'adverbe comme *vraiment beaucoup* : en effet, *vraiment* agit sur *beaucoup* et est donc un transformateur de transformateur d'application $D^2 \to \{\text{vrai}, \text{faux}\}$...

Comment gérer cette complexité qui semble croître inexorablement ?

Voici une modélisation mathématique qui nous délivre du cauchemar décrit ci-dessus : considérons qu'il n'existe que deux *types sémantiques primitifs* : celui de «formule» (dont l'interprétation dans D sera «vrai» ou «faux») et celui de «valeur» (dont l'interprétation sera un élément du domaine D). Notons t les formules et e les valeurs. Ensuite prenons le monoïde libre $\{e, t\}^*$ dont nous notons la loi comme un produit scalaire $\langle , \rangle$ [1].

Appelons les éléments de ce monoïde des *types sémantiques complexes*. Nous allons faire *correspondre chaque sommet de l'arbre syntaxique à un type sémantique complexe*, et ceci de la manière suivante : l'élément de gauche de $\langle , \rangle$ est la «donnée d'entrée» du type, son élément de droite est sa «donnée de sortie».

Exemple : un prédicat unaire, comme *dort*, s'applique à une constante, son interprétation fournit une valeur de $\{\text{vrai}, \text{faux}\}$. On dira donc qu'il est de type $\langle e, t\rangle$ (= «il prend un e et il nous rend un t»).

Mais attention : on ne prend qu'un seul élément d'entrée à la fois ! Un prédicat *binaire* ne sera donc pas de type $\langle \{e, e\}, t\rangle$ (cette syntaxe n'est pas valide) mais sera décrit de *manière récursive* comme $\langle e, \langle e, t\rangle\rangle$.

Si on y réfléchit un peu, c'est parfaitement logique : un prédicat binaire auquel on fournit une valeur devient prédicat unaire, donc pour une entrée e, la sortie est $\langle e, t\rangle$. De même, le type d'un prédicat ternaire sera $\langle e, \langle e, \langle e, t\rangle\rangle\rangle$ et ainsi de suite...

De la même manière, la particule de conjonction *et* sera de type $\langle t, \langle t, t\rangle\rangle$, l'adverbe *beaucoup* de type $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$, et le modificateur d'adverbe *vraiment*, de type $\langle\langle\langle e, t\rangle, \langle e, t\rangle\rangle, \langle\langle e, t\rangle, \langle e, t\rangle\rangle\rangle$. Arrivé à ce stade, le lecteur/la lectrice doit normalement se sentir ébloui(e) devant l'époustouflante beauté de ce modèle : en effet, *quelque soit* le type sémantique d'un mot, aussi complexe soit-il, on peut le décrire simplement par un élément du monoïde libre $\{e, t\}^*$... c'est simple et efficace.

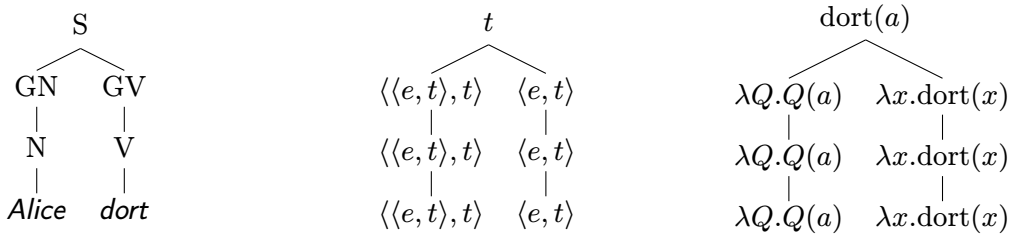

FIGURE 1 – Arbre syntaxique de la phrase *Alice dort*, arbre de ses types sémantiques, arbre de la traduction en formalisme logique.

Sur la fig. 1 le lecteur peut constater que
- à chaque fois qu'un sommet n'a qu'un seul enfant, le type sémantique ne change pas, et
- à chaque fois qu'un sommet a plusieurs enfants, leurs types sémantiques se *composent* : ainsi, $\langle\langle e, t\rangle, t\rangle$ appliqué à $\langle e, t\rangle$ donne t. On note $\times$ cette composition : $\langle e, t\rangle \times e = t$. Pour qu'une composition $T_1 \times T_2$ puisse avoir lieu, il faut que T_1 soit un type complexe et que sa première composante soit égale à T_2.

La cohérence sémantique d'une phrase provient du fait que les types sémantiques des sommets de son arbre syntaxique se composent correctement, pour arriver au type de S qui est, invariablement, t (comme on peut le constater sur le graphe de la fig. 1).

NLTK peut nous calculer les types à partir des formules logiques. En écrivant

```
from nltk.sem.logic import *
tlp = LogicParser(True)
print(tlp.parse(r'\Q.Q(alice)').type)
print(tlp.parse(r'\x.dort(x)').type)
print(tlp.parse(r'\Q.Q(alice) (\x.dort(x))').type)
```

on a le retour

1. Attention : cette loi *n'est pas* associative, donc pas question de faire des «simplifications» : $\langle e, \langle e, t\rangle\rangle$ n'est pas la même chose que $\langle\langle e, e\rangle, t\rangle$!

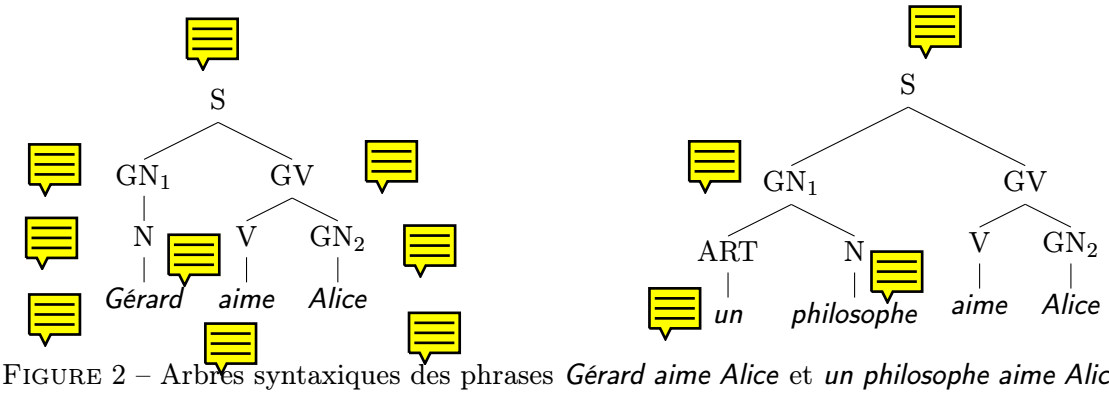

FIGURE 2 – Arbres syntaxiques des phrases *Gérard aime Alice* et *un philosophe aime Alice*.

```
<<e,?>,?>
<e,?>
?
```

Le point d'interrogation vient du fait que le parseur ne sait pas si «Q» et «dort» sont des prédicats (et donc de type t) ou des fonctions (type e). Autre fonctionnalité intéressante : NLTK peut faire des β-réductions pour nous. Ainsi, si on se demande que peut bien signifier $\lambda Q.Q(\text{alice})(\lambda x.\text{dort}(x))$, il suffit d'écrire

```
from nltk.sem.logic import *
lexpr = Expression.fromstring
print(lexpr(r'\Q.Q(alice) (\x.dort(x))').simplify())
```

et on a la réponse :

```
dort(alice)
```

On peut ainsi vérifier et nos types et nos réductions.

2.1.1 Exercice sur l'amour

En se basant sur les résultat du TP 1, calculer l'arbre des types de la phrase *Gérard aime Alice* (fig. 2). Vérifier sous Python NLTK.

2.2 L'article indéfini

Prenons la phrase *un philosophe aime Alice* (fig. 2). Notons tout de suite que *philosophe* ne peut être traduit par une constante, comme, par exemple *Gérard*, puisque «être philosophe» est une propriété, et les propriétés sont naturellement traduites par des prédicats unaires. Ainsi, on écrira philosophe(g) pour dire que g est philosophe. De même, $\exists x$ philosophe(x) signifie qu'il existe un philosophe. Et donc, sa traduction en formalisme logique est $\exists x\,(\text{philosophe}(x) \wedge \text{aime}(x, a))$.

En donner l'arbre des types et l'arbre des formalisations logiques. Vérifier sous NLTK.

2.3 L'article défini

Mais comment traduire alors l'article défini *le*, dans la phrase *le philosophe aime Alice* ?

Formulons la question autrement : comment indiquer qu'il n'y a qu'*un seul* philosophe, et que quand on dit *le philosophe* on parle justement de lui ?

Voici comment décrire l'*unicité* : il n'y a qu'un seul philosophe x si et seulement si pour tout individu y tel que y soit philosophe, on ait $x = y$.

La phrase *le philosophe aime Alice* se traduira donc par $\exists x\,(\forall y\,(\text{philosophe}(y) \leftrightarrow x = y) \wedge \text{aime}(x, a))$.

En donner l'arbre des types et l'arbre des formalisations logiques. Vérifier sous NLTK. Comment justifier le fait qu'un mot aussi simple et aussi élémentaire a une traduction en formalisme logique aussi complexe ?