Lambda-calcul

Introduction à la sémantique formelle

David Blunier · Université de Poitiers L3 · Printemps 2025

Vers un système compositionnel

- Rappelez-vous de notre introduction que notre but est de développer une sémantique compositionnelle pour les langues naturelles.
- Une sémantique compositionnelle est un système formel permettant d'attribuer une signification à chaque constituant atomique de la phrase, qui reste la même quelle que soit la complexité de la structure dans laquelle il est utilisé.

Vers un système compositionnel

- Aujourd'hui nous allons apprendre à maîtriser l'outil formel le plus puissant à cette fin: le **lambda-calcul**.
- Le lambda-calcul va nous permettre de développer un nouveau langage, L-lambda (LL), basé sur LPred.
- Ce nouveau langage nous permettra de créer un système (quasi-)complètement compositionnel.

Le problème avec LPred

- Considérons une phrase simple du français:

$$[_{NP}\ \text{Selina}\ [_{VP}\ [_{V}\ \text{aime}]\ [_{NP}\ \text{Fido}]]]$$

- Grâce à LPred, nous pouvons assigner une dénotation aux constituants suivants:

> $[\![s]\!]^{M,g}$ = Selina
> $[\![f]\!]^{M,g}$ = Fido
> $[\![aime(s, f)]\!]^{M,g}$ = Selina aime Fido

Le problème avec LPred

- Problème: comment faisons-nous pour représenter le syntagme verbal VP *aime Fido*?

$$[_{NP} \text{ Selina } [_{VP} \; [_{V} \text{ aime}] \; [_{NP} \text{ Fido}]]]$$

Le problème avec LPred

- Nous ne pouvons pas écrire la chose suivante, car ce n'est pas une formule bien formée de LPred:

> 💀 $[\![aime(x, f)]\!]^{M,g}$ = ?

Même si la fonction d'assignation g était définie pour x, alors cette expression serait équivalente à

> 💀 $[\![\exists x.aime(x, f)]\!]^{M,g}$

Or, la signification de ceci est *quelqu'un aime Fido*, non pas *aime Fido*!

Le problème avec LPred

- Intuitivement, on aimerait représenter *aime Fido* d'une façon analogue à *aime*, i.e. comme un prédicat binaire dont seul un argument a été saturé:

$$aime(\quad, f)$$

- Or, c'est exactement ce que le lambda-calcul va nous permettre de faire grâce à l'introduction d'un nouvel outil: **la fonction lambda**.

La fonction lambda

- La **fonction lambda**, également appelée **abstracteur lambda** ou **lambda-abstracteur**, est un élément nous permettant d'introduire une variable en tant qu'argument d'un prédicat "non-saturé", i.e. vide, afin de la lier.

$$\lambda x.aime(x, f)$$

- Comme toutes les variables doivent être liées (dans LL comme dans LPred), le résultat est une structure interprétable!
- La formule finale ici peut se lire "la fonction caractéristique de l'ensemble des éléments (dans M) qui aiment Fido", i.e., n'importe quel x tel que x aime Fido, ce qui est (intuitivement) la signification de notre VP!

La fonction lambda

$$\lambda x.aime(x, f)$$

- Plus techniquement, cette formule dénote une **fonction d'un individu à une valeur de vérité**, une fonction dont la valeur sera de 1(vraie) ssi cet individu aime Fido.
- C'est ce que l'on appelle la "fonction caractéristique de l'ensemble" de tous les individus qui aiment Fido.

La fonction lambda (λ)

- À l'aide de λ, nous pouvons également exprimer la signification d'éléments qui semblent ne pas avoir de signification autre que logique (i.e., ne renvoient ni à des individus, ni à des prédicats), comme *tout*.
- Intuitivement, la dénotation de *tout* semble être ceci:

$$[\![tout]\!]^{M,g} = \forall x. \quad (x)$$

Où l'espace blanc indique que cette position doit être remplie par un prédicat.

La fonction lambda (λ)

- Nous pouvons capturer la signification de *tout* en effectuant une lambda-abstraction sur des **prédicats**.
- Voici par exemple l'entrée lexicale (= la dénotation) de *tout* (adverbe):

$$[\![tout]\!]^{M,g} = \lambda P.\lambda Q.\forall(x).[P(x) \rightarrow Q(x)]$$

"La fonction caractéristique de l'esemble des prédicats P et Q telle que pour tout x, si $P(x)$, alors $Q(x)$".

La fonction lambda (λ)

- Ceci est une nouveauté de LL par rapport à LPred:
- Dans LPred, nous n'avions que des **variables d'individus** $x, y, z...$
- Dans LL, nous avons également des **variables de prédicats** $P, Q...$
- Ceci nous permet d'identifier LL comme une logique **d'ordre supérieur (*higher-order logic*)**.

Quelques applications

- Grâce à notre nouvel outil, nous pouvons représenter beaucoup d'expressions complexes des langues naturelles. Voici quelques exemples:

> $[\![\text{vapoteur}]\!]^{M,g} = \lambda x.vapote(x)$ = l'ensemble des individus qui vapotent.
> $[\![\text{non-vapoteur}]\!]^{M,g} = \lambda x.\neg vapote(x)$ = les non-vapoteurs.
> $[\![\text{tous}]\!]^{M,g} = \lambda P.\lambda Q.\forall x.[P(x) \rightarrow Q(x)]$ = l'ensemble des choses possédant les propriétés P et Q.
> $[\![\text{tous les vapoteurs}]\!]^{M,g} = \lambda Q.\forall x.vapote(x) \rightarrow Q(x)$ = tous les vapoteurs.
> $[\![\text{tous les vapoteurs sont dehors}]\!]^{M,g} = \forall x.vapote(x) \rightarrow dehors(x)$

Corps de la fonction λ

- Dans toute expression de la forme $\lambda x.\phi$, la partie ϕ désigne la **portée** de l'expression λ, c'est-à-dire la valeur de la fonction étant donné un argument: c'est ce que l'on appelle le **corps de la fonction**.
- Ansi, dans l'expression

$$\lambda x.aime(s, x)$$

le corps de la fonction est $aime(s, x)$, l'ensemble des individus aimés par Selina.

Exercice

- Identifiez le corps des fonction suivantes:

> $\lambda x.sympa(x)$
> $\lambda x.x$
> $\lambda y.\lambda x.[aime(x,y) \vee aime(y,x)]$
> $\lambda z.\lambda y.\lambda x.entre(x,y,z)$

Lambda-conversion (beta-réduction)

- Tout comme dans LPred, nous indiquons l'argument auquel la fonction s'applique entre parenthèses, à droite. Ainsi

$$[\lambda x.aime(x, f)](s)$$

Dans notre modèle équivaut à appliquer la fonction *aime Fido* à l'individu Selina.

- Une fois la fonction appliquée, cette formule devient équivalente à:

$$aime(s, f)$$

- On appelle **lambda-conversion (λ-conversion, également beta-réduction)** l'application de cette fonction.

Lambda-conversion (beta-réduction)

> **λ-conversion**
>
> Pour toute variable x et expression α,
> $[\lambda x....x...](\alpha) \equiv \alpha$

- Ainsi, les expressions suivantes sont équivalentes:

> $[\lambda x.heureux(x)](s) \equiv heureux(s)$
> $[\lambda x.[heureux(x) \wedge chat(x)]](s) \equiv heureux(s) \wedge chat(s)$

Exercice

- Lorsque c'est possible, appliquez une λ-conversion aux fonction suivantes:

> $[\lambda x.x](f)$
> $[\lambda P.P](homme)$
> $[\lambda x.P(x)](f)$
> $[\lambda y.\lambda x.R(y,x)](f)$
> $[\lambda x.R(y,s)](f)$
> $[\lambda P.\exists x.P(x)](homme)$
> $[\lambda P.\forall x.P(x)](mortel)$

Exercice

- Lorsque c'est possible, appliquez une λ-conversion aux fonction suivantes:

> $[\lambda x.x](f) \equiv f$
> $[\lambda P.P](homme) \equiv homme$
> $[\lambda x.P(x)](f) \equiv P(f)$
> $[\lambda y.\lambda x.R(y,x)](f) \equiv \lambda x.R(f,x)$ (*via* alpha-conversion)
> $[\lambda x.R(y,s)](f)$: pas de réduction possible (la variable x n'est pas liée)
> $[\lambda P.\exists x.P(x)](homme) \equiv \exists x.homme(x)$
> $[\lambda P.\forall x.P(x)](mortel) \equiv \forall x.mortel(x)$

Alpha-conversion

- De façon importante, la lambda-conversion ne peut saturer **que les variables liées par l'opérateur λ**, et non pas les variables déjà liées par un autre opérateur.
- Par conséquent, les formules suivantes sont équivalentes:

> $[\lambda x.[sourire(x) \wedge \exists x.heureux(x)]](a) \equiv$
> $[sourire(a) \wedge \exists x.heureux(x)]]$

- **Notez bien que la seconde occurence de x n'est pas saturée par a**, puisque cette variable est liée par l'opérateur $\exists$.

Alpha-conversion

- Bien que rien n'empêche une même variable d'être liée par plusieurs opérateurs dans LL (en réalité, les opérateurs ne lient que des **occurences d'une variable**), la récurrence des variables peut porter à confusion.
- C'est pourquoi nous pouvons adopter la **conversion alpha**, qui est une règle de réécriture: **toute variable liée peut être remplacée par une autre variable de catégorie similaire sans changer la formule.**
- Ainsi, les formules suivantes sont équivalentes, où nous avons remplacé toutes les instances de x liées par $\forall$ par y:

> $\forall x.P(x) \equiv$
> $\forall y.P(y)$

Alpha-conversion

- Les formules suivantes sont équivalentes par la même règle:

> $\lambda x.P(x) \equiv$
> $\lambda y.P(y)$

> $\lambda x.aime(x, y) \equiv$
> $\lambda z.aime(z, y)$

Alpha-conversion

- Dans certains cas, la conversion alpha est nécessaire:

$$[\lambda x.\lambda y.aime(x,y)](x)$$

- Si nous appliquons la conversion lambda, nous obtenons:

$$\lambda x.aime(x,x)]$$

- **Mais ces deux formules ne sont pas équivalentes!**
- Il s'agit d'un cas de **capture de variable accidentelle**, également appelé **collision**: la substitution de y pour x a entraîné une identification des deux variables.

Alpha-conversion

- La conversion α nous permet d'éviter ceci, en réécrivant

$$[\lambda x.\lambda y.aime(x,y)](x)$$

De la façon suivante

$$[\lambda y.\lambda z.aime(z,y)](x)$$

Où toutes les occurences de y liées par le premier λ ont été substituées par z.

- Nous obtenons ensuite (par lambda conversion et saturation de y par x):

$$\lambda z.aime(z,x)$$

Exercice

- Appliquez la règle de conversion α à au moins un variable contenue dans les expressions suivantes:

> $\lambda x.mignon(x)$
> $\lambda x.x$
> $\lambda y.\lambda x.[aime(x,y) \lor aime(y,x)]$
> $\lambda z.\lambda y.\lambda x.entre(x,y,z)$
> $\lambda x.[chien(x) \land \exists x.faim(x)]$