Semantic Theory
Week 3 – Typed Lambda Calculus

Noortje Venhuizen
Harm Brouwer

Universität des Saarlandes

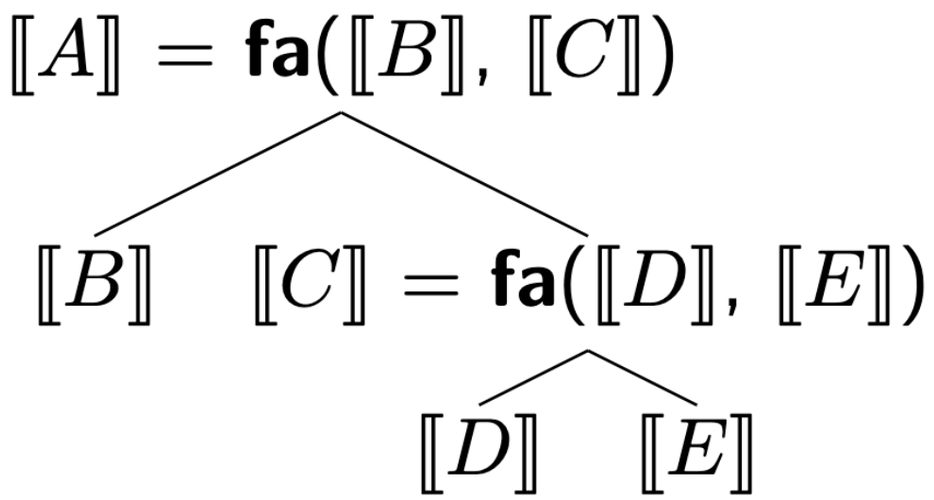
Summer 2021

Compositionality

The principle of compositionality: "The meaning of a complex expression is a function of the meanings of its parts and of the syntactic rules by which they are combined" (Partee et al.,1993)

Compositional semantics construction:

1. Compute meaning representations for sub-expressions
2. Combine them in a principled manner to obtain a meaning representation for a complex expression.



Compositionality: Example

Bill is not a student => [NOT [[bill]$_{NP}$ [student]$_{VP}$]$_S$]$_S$

$[\![\neg\text{student'(bill')}]\!] \in D_t : 0$

$[\![\text{student'(bill')}]\!] \in D_t : 1$

$[\![\text{NOT}]\!] \in D_{\langle t,t\rangle}$
$= \neg$

$\begin{bmatrix} 1 \rightarrow 0 \\ 0 \rightarrow 1 \end{bmatrix}$

$[\![\text{bill'}]\!] \in D_e$
$= V_M(\text{bill'})$

e_1

$[\![\text{student'}]\!] \in D_{\langle e,t\rangle}$
$= V_M(\text{student'})$

$\begin{bmatrix} e_1 \rightarrow 1 \\ e_2 \rightarrow 0 \\ e_3 \rightarrow 1 \\ e_4 \rightarrow 1 \\ e_5 \rightarrow 0 \end{bmatrix}$

Explicating Functions and Arguments

Bill is not a student => [NOT [[bill]$_{NP}$ [student]$_{VP}$]$_S$]$_S$

$[\![\lambda S_t.\neg S(\text{student'}(\text{bill'}))]\!] = [\![\neg \text{student'}(\text{bill'})]\!] \in D_t$

$[\![\lambda x_e.\text{student'}(x)(\text{bill'})]\!] =$
$[\![\text{student'}(\text{bill'})]\!] \in D_t$

$[\![\text{NOT}]\!] \in D_{\langle t,t\rangle}$
$= [\![\lambda S_t.\neg S]\!]$

$[\![\text{bill'}]\!] \in D_e$

$[\![\text{student'}]\!] \in D_{\langle e,t\rangle}$
$= [\![\lambda x_e.\text{student'}(x)]\!]$

Expressiveness of lambda functions

- Lambda expressions allow for explicitly differentiating between Functions and Arguments

- Functions consist of a body and a set of lambda variables

- Body of Functions can contain logical operators

- Functions can themselves serve as Arguments

$$[\ [\text{Not smoking}_{\langle e,t\rangle}]\ [\text{is healthy}]_{\langle\langle e,t\rangle,t\rangle}]$$

$$[\![\text{healthy'}(\lambda y_e.\neg(\text{smoking}_{\langle e,t\rangle}(y)))]\!] \in D_t$$

Lambda abstraction

Winter-EFS Ch3
Page 65

λ-abstraction is the operation that transforms expressions of any type τ into a function ⟨σ,τ⟩, where σ is the type of the λ-variable.

Formal definition:

If α is in WE_{τ}, and x is in VAR_{σ} then λx(α) is in $WE_{\langle\sigma,\,\tau\rangle}$

- The scope of the λ-operator is the smallest WE to its right. Wider scope must be indicated by brackets.
- We often use the “dot notation” λx.ϕ indicating that the λ-operator takes widest possible scope (over ϕ).

Interpretation of Lambda-expressions

If $\boldsymbol{\alpha} \in WE_\tau$ and $v \in VAR_\sigma$, then $[\![\lambda v \boldsymbol{\alpha}]\!]^{M,g}$ is that function $f : D_\sigma \rightarrow D_\tau$ such that for all $a \in D_\sigma$, $f(a) = [\![\boldsymbol{\alpha}]\!]^{M,g[v/a]}$

If the λ-expression is applied to some argument, we can simplify the interpretation:

- $[\![\lambda v \boldsymbol{\alpha}]\!]^{M,g}([\![x]\!]^{M,g}) = [\![\boldsymbol{\alpha}]\!]^{M,g[v/[\![x]\!]^{M,g}]}$

Example: *"Bill is a student"*

$[\![\lambda x(S(x))(b')]\!]^{M,g} = 1$

iff $[\![\lambda x(S(x))]\!]^{M,g}([\![b']\!]^{M,g}) = 1$

iff $[\![S(x)]\!]^{M,g'} = 1$ where $g'=g[x/[\![b']\!]^{M,g}]$

iff $[\![S]\!]^{M,g'}([\![x]\!]^{M,g'}) = 1$

iff $V_M(S)(V_M(b')) = 1$

Hence:

$[\![\lambda x(S(x))(b')]\!]^{M,g} = [\![S(b')]\!]^{M,g}$

- We can use this equivalence to simplify lambda expressions, using an operation called β-reduction

β-Reduction (Function application)

Winter-EFS Ch3
Page 67

$$[\![\lambda v(\boldsymbol{\alpha})(\boldsymbol{\beta})]\!]^{M,g} = [\![\boldsymbol{\alpha}]\!]^{M,g[v/[\![\beta]\!]M,g]}$$

⇒ all (free) occurrences of the λ-variable in **α** get the interpretation of **β** as value.

This operation is called β-reduction

- **λv**(**α**)(**β**) ⇔ **α**[**β**/**v**]
- **α**[**β**/**v**] is the result of replacing all free occurrences of **v** in **α** with **β**

Achtung: The equivalence is not unconditionally valid …

Variable capturing

Q: Are λv(α)(β) and α[β/v] always equivalent?

- λx(drive'(x) ∧ drink'(x))(j') ⇔ drive'(j') ∧ drink'(j')
- λx(drive'(x) ∧ drink'(x))(y) ⇔ drive'(y) ∧ drink'(y)
- λx(∀y know'(x)(y))(j') ⇔ ∀y know(j')(y)
- λx(∀y know'(x)(y))(y) ⇏ ∀y know(y)(y) Problem: y is not "free for x"

Definition: Let v, v' be variables of the same type, and let **α** be any well-formed expression.

- v is free for v' in **α** iff no free occurrence of v' in **α** is in the scope of a quantifier or a λ-operator that binds v.

Conversion rules

- β-conversion: $\lambda v(\boldsymbol{\alpha})(\boldsymbol{\beta}) \Leftrightarrow \boldsymbol{\alpha}[\boldsymbol{\beta}/v]$

  (if all free variables in **β** are free for v in **α**)

- α-conversion: $\lambda v.\boldsymbol{\alpha} \Leftrightarrow \lambda w.\boldsymbol{\alpha}[w/v]$

  (if w is free for v in **α**)

- η-conversion: $\lambda v.\boldsymbol{\alpha}(v) \Leftrightarrow \boldsymbol{\alpha}$

Determiners as lambda-expressions

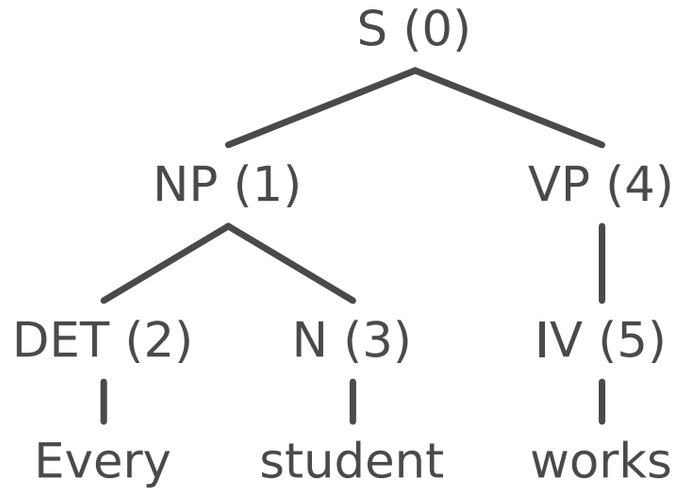
- a student works ➔ ∃x(student'(x) ∧ work'(x)) :: t
  - a student ➔ λP∃x(student'(x) ∧ P(x)) :: ⟨⟨e,t⟩,t⟩
  - a, some ➔ λQλP∃x(Q(x) ∧ P(x)) :: ⟨⟨e,t⟩,⟨⟨e,t⟩,t⟩⟩
- every student ➔ λP∀x(student'(x) → P(x)) :: ⟨⟨e,t⟩,t⟩
  - every ➔ λQλP∀x(Q(x) → P(x)) :: ⟨⟨e,t⟩,⟨⟨e,t⟩,t⟩⟩
- no student ➔ λP¬∃x(student(x) ∧ P(x)) :: ⟨⟨e,t⟩,t⟩
  - no ➔ λQλP¬∃x(Q(x) ∧ P(x)) :: ⟨⟨e,t⟩,⟨⟨e,t⟩,t⟩⟩
- someone ➔ λF∃xF(x) :: ⟨⟨e,t⟩,t⟩

β-Reduction Example

Every student works.

(2) λPλQ∀x(P(x) → Q(x)) :: ⟨⟨e, t⟩, ⟨⟨e, t⟩, t⟩⟩

(3) λx.student'(x) ⇔η student' :: ⟨e, t⟩



(1) λPλQ∀x(P(x) → Q(x))(student')
⇔β λQ∀x(student'(x) → Q(x)) :: ⟨⟨e, t⟩, t⟩

(4)/(5) λx.work'(x) ⇔η work' :: ⟨e, t⟩

(0) λQ∀x(student'(x) → Q(x))(work') ⇔β ∀x(student'(x) → work'(x)) :: t

Compositionally deriving complex expressions

Not smoking is healthy => [[Not smoking] [is healthy]]

$[\![\lambda P_{\langle e,t\rangle}.\text{healthy'}(P)(\lambda y_e.\neg\text{smoking}(y))]\!]$
$\Leftrightarrow^{\beta} [\![\text{healthy'}(\lambda y_e.\neg\text{smoking}(y))]\!] \in D_t$

$[\![\text{NOT smoking}]\!] \in D_{\langle e,t\rangle}$
$= [\![\lambda P_{\langle e,t\rangle}\lambda y_e.\neg P(y)(\lambda x_e.\text{smoking}(x))]\!]$
$\Leftrightarrow^{\beta} [\![\lambda y_e.\neg(\lambda x_e.\text{smoking}(x))(y)]\!]$
$\Leftrightarrow^{\beta} [\![\lambda y_e.\neg\text{smoking}(y)]\!]$

$[\![\text{NOT}]\!] \in D_{\langle\langle e,t\rangle,\langle e,t\rangle\rangle}$
$= [\![\lambda P_{\langle e,t\rangle}\lambda x_e.\neg P(x)]\!]$
$\Leftrightarrow^{\alpha} [\![\lambda P_{\langle e,t\rangle}\lambda y_e.\neg P(y)]\!]$

$[\![\text{smoking'}]\!] \in D_{\langle e,t\rangle}$
$= [\![\lambda x_e.\text{smoking}(x)]\!]$

$[\![\text{healthy'}]\!] \in D_{\langle\langle e,t\rangle,t\rangle}$
$= [\![\lambda P_{\langle e,t\rangle}.\text{healthy'}(P)]\!]$

Transitive Verbs: Type Clash

- Someone reads a book

$$\frac{\text{read} :: \langle e,\langle e, t\rangle\rangle \qquad \text{a book} :: \langle\langle e, t\rangle,t\rangle}{\dfrac{\text{someone} :: \langle\langle e, t\rangle,t\rangle \qquad ?? :: ??}{?? :: t}}$$

Solution: reverse functor-argument relation (again)

- Logical form: someone(read(a book))
- Adjust type of first argument of transitive verb: $\text{read}_{\langle\langle\langle e, t\rangle,t\rangle,\langle e, t\rangle\rangle}$ (*Type Raising*)

Type Raising

It's not enough to just change the type of the transitive verb:

- read ➔ read' ∈ $CON_{\langle\langle e,t\rangle, t\rangle, \langle e, t\rangle\rangle}$

  someone reads a book:
  λFꓱxF(x)(read'(λPꓱy(book'(y) ∧ P(y)))
  $\Leftrightarrow^{\beta}$ ꓱx(read'(λPꓱy(book'(y) ∧ P(y)))(x) … No further reduction steps possible.

  Problem: this does not support the following entailment:
  someone reads a book ⊨ *there exists a book*

Hence, we need a more explicit λ-term:

- read ➔ λ*Q*λz.*Q*(λx(read*(x)(z))) ∈ $WE_{\langle\langle e,t\rangle, t\rangle, \langle e, t\rangle\rangle}$
  where: read* ∈ $WE_{\langle e, \langle e, t\rangle\rangle}$ is the "underlying" first-order relation

Transitive Verbs: example

someone reads a book: someone(read(a book))

λFꓱxF(x)(λQλz(Q(λx(read(x)(z))))(λRλP(ꓱy(R(y) ∧ P(y)))(book'))))*

⇔β λFꓱxF(x)(λQλz(Q(λx(read*(x)(z))))(λP(ꓱy(book'(y) ∧ P(y))))))

⇔β λFꓱxF(x)(λz(λP(ꓱy(book'(y) ∧ P(y)))(λx(read*(x)(z)))))

⇔β λFꓱxF(x)(λz(ꓱy(book'(y) ∧ λx(read*(x)(z))(y))))

⇔β λFꓱxF(x)(λz(ꓱy(book'(y) ∧ read*(y)(z))))

⇔β ꓱx(λz(ꓱy(book'(y) ∧ read*(y)(z)))(x))

⇔β ꓱxꓱy(book'(y) ∧ read*(y)(x))

Reading material

- Winter: Elements of Formal Semantics (Chapter 3, Part III) http://www.phil.uu.nl/~yoad/efs/main.html