Semantic Theory

Lecture 3:
Elementary semantics construction

M. Pinkal / A. Koller
Summer 2006

Last week: Type theory

- Expressive limits of FOL
  - John is a blond/honest/alleged criminal.
  - Bill is blond. Blond is a hair colour.
- Solution: Generalise FOL to type theory
  - basic types e, t
  - functional types $\langle\sigma, \tau\rangle$
  - build logic from functional application and the usual logical connectives (over higher-order constants and variables).

Building well-formed expressions

Bill drives fast.

drive: <e,t> fast: <<e,t>,<e,t>>

Bill: e fast(drive): <e,t>

fast(drive)(bill): t

Mary works in Saarbrücken

mary: e work: <e,t> in: <e,<t,t>> sb: e

work(mary): t in(sb): <t,t>

in(sb)(work(mary)): t

Type-theoretic semantics [1]

- Let U be a non-empty set of entities.
- The domain of possible denotations D_τ for every type τ is given by:
  - $D_e = U$
  - $D_t = \{0,1\}$
  - $D_{<\sigma, \tau>}$ is the set of all functions from D_σ to D_τ

Type-theoretic semantics [2]

- A model structure for a type theoretic language:
  M = <U, V>, where
  - U (or U_M) is a non-empty domain of individuals
  - V (or V_M) is an interpretation function, which assigns to every member of Con_τ an element of D_τ.
- Variable assignment g assigns every variable of type τ a member of D_τ.

Type-theoretic semantics [3]

Interpretation (with respect to model structure M and variable assignment g):

$[[\alpha]]^{M,g} = V_M(\alpha)$, if α constant

$[[\alpha]]^{M,g} = g(\alpha)$, if α variable

$[[\alpha(\beta)]]^{M,g} = [[\alpha]]^{M,g}([[\beta]]^{M,g})$

$[[\neg\varphi]]^{M,g} = 1$ iff $[[\varphi]]^{M,g} = 0$

$[[\varphi \wedge \psi]]^{M,g} = 1$ iff $[[\varphi]]^{M,g} = 1$ and $[[\psi]]^{M,g} = 1$, etc.

If $v \in Var_\tau$, $[[\exists v\varphi]]^{M,g} = 1$ iff there is $a \in D_\tau$ such that $[[\varphi]]^{M,g[v/a]} = 1$

If $v \in Var_\tau$, $[[\forall v\varphi]]^{M,g} = 1$ iff for all $a \in D_\tau$: $[[\varphi]]^{M,g[v/a]} = 1$

$[[\alpha=\beta]]^{M,g} = 1$ iff $[[\alpha]]^{M,g} = [[\beta]]^{M,g}$

Sets as characteristic functions

- One trick we've applied a lot was to model sets by their characteristic functions.
- A function of type <σ, t> maps each member of D_σ to true or false.
- See this as representing a subset of D_σ (namely, the set of members of D_σ that are mapped to true).
- Example: "blond" is a constant of type <e,t>. It can be seen as characterising the set of blond individuals (of type e).

Outline

- Elementary semantics construction:
  - the principle of compositionality
  - compositional semantics construction using type theory
- Quantified noun phrases: A challenge for compositionality
- The lambda operator in type theory.
- Things work!

Frege's Principle

... or the Principle of Compositionality:

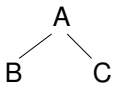
- The meaning of a complex expression is uniquely determined by the meanings of its sub-expressions and its syntactic structure.

Two levels of interpretation

- Semantic interpretation of a NL expression in a logical framework is a two-step process:
  - The NL expression is assigned a semantic representation
  - The semantic representation is truth-conditionally interpreted.
- Truth-conditional interpretation of logical representations is strictly compositional.
- We also want this for the process of computing logical representations from NL expressions.

Some basic rules

- Rule of functional application:



$$\frac{B \Rightarrow \beta: \langle\sigma, \tau\rangle \quad C \Rightarrow \gamma:\sigma}{A \Rightarrow \beta(\gamma): \tau} \quad \text{or} \quad \frac{B \Rightarrow \beta: \sigma \quad C \Rightarrow \gamma: \langle\sigma, \tau\rangle}{A \Rightarrow \gamma(\beta): \tau}$$

- Rule of non-branching nodes:

$$\frac{B \Rightarrow \beta: \tau}{A \Rightarrow \beta: \tau}$$

Some basic rules

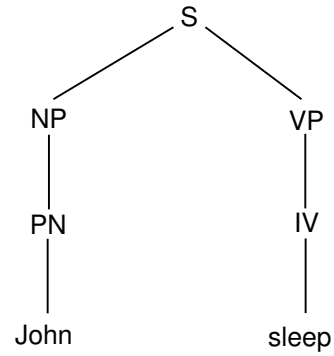
- Rule of lexical nodes:

$$\frac{}{A \Rightarrow \beta: \tau}$$

The semantic representation β for the word "a" is supplied by the lexicon.

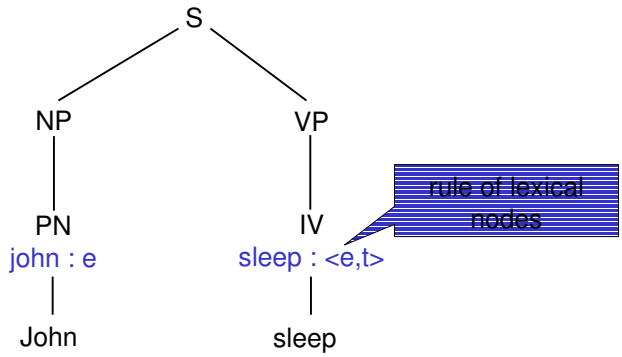
Semantics construction

- John sleeps.



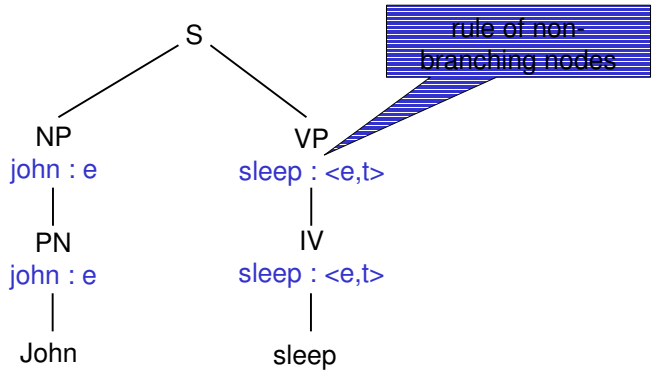
Semantics construction

- John sleeps.



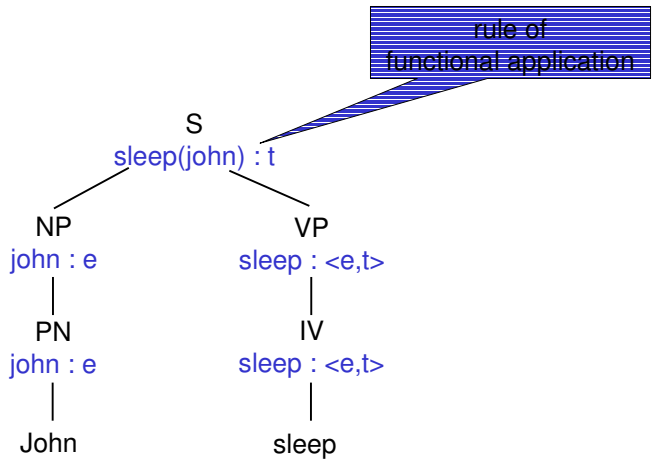
Semantics construction

- John sleeps.



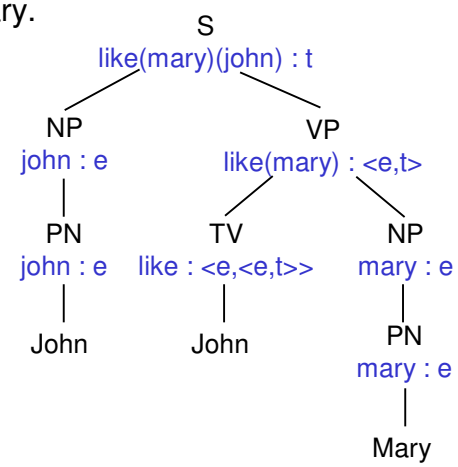
Semantics construction

- John sleeps.



Semantics construction

- John likes Mary.

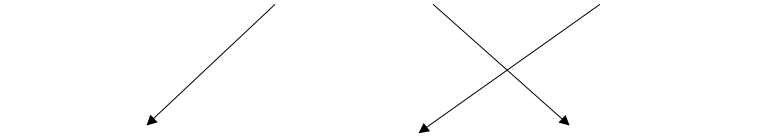


Semantics construction

- This looks pretty neat!
- In fact, we will keep the same semantics construction rules for a little while.
- But ...

Noun phrases and compositionality

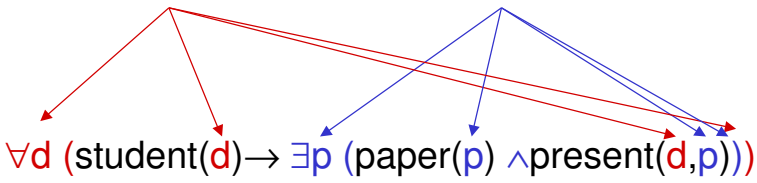
Every student presented a paper



$\forall d\ (student(d) \rightarrow \exists p\ (paper(p) \wedge present(d,p)))$

Noun phrases and compositionality

Every student presented a paper



$\forall d\ (student(d) \rightarrow \exists p\ (paper(p) \wedge present(d,p)))$

Noun phrases and compositionality

John works.	*work(john)*
Somebody works	∃x (work(x))
Every student works	∀x (student(x)→ work(x))
No student works	¬ ∃x (student(x) ∧ work(x))
John and Mary work	*work(john)* ∧ work(mary)

What's the semantic representation of a noun phrase?

Towards a unified semantics of NPs

John works.

john: e work: <e,t>

work(john): t

Every student works.

every-student: <<e,t>,t> work: <e,t>

every-student(work): t

Towards a unified semantics of NPs

John works.

??? : <<e,t>,t> work: <e,t>

john(work): t

Every student works.

every-student: <<e,t>,t> work: <e,t>

every-student(work): t

A coverage problem

Swimming is healthy

swimming: <e,t> healthy <<e,t>,t>

healthy(swimming): t

Not smoking is healthy

Driving and drinking is dangerous

John drives and drinks

Some people drive and drink

Summing up the problems

- We have the following kinds of problems:
  - We want uniform semantic representations for noun phrases, and we don't seem to have the syntax to write them down.
  - Some NL expressions seem to require us to say "an x with the property P".
- So, let's extend the logic.

The solution: λ-abstraction

$\lambda x[drive(x) \wedge drink(x)]$

- a term of type <e,t>
- denotes the property of being "an x such that x drives and drinks"
- λ-abstraction is an operation that takes an expression and „opens“ or „re-opens“ specific argument positions by abstracting over a variable
- The result of abstraction over individual variable x in the formula $drive(x) \wedge drink(x)$ results in the complex predicate $\lambda x[drive(x) \wedge drink(x)]$.

Syntax of λ-abstraction

If $\alpha \in WE_{\tau}$, $v \in Var_{\sigma}$, then $\lambda v \alpha \in WE_{<\sigma,\tau>}$.

Notational convention:
The scope of the λ-operator is the smallest WE to its right. Wider scope must be indicated by brackets.

Example

drive: <e,t> x:e drink: <e,t> x:e

drive(x): t drink(x): t

drive(x)∧drink(x): t

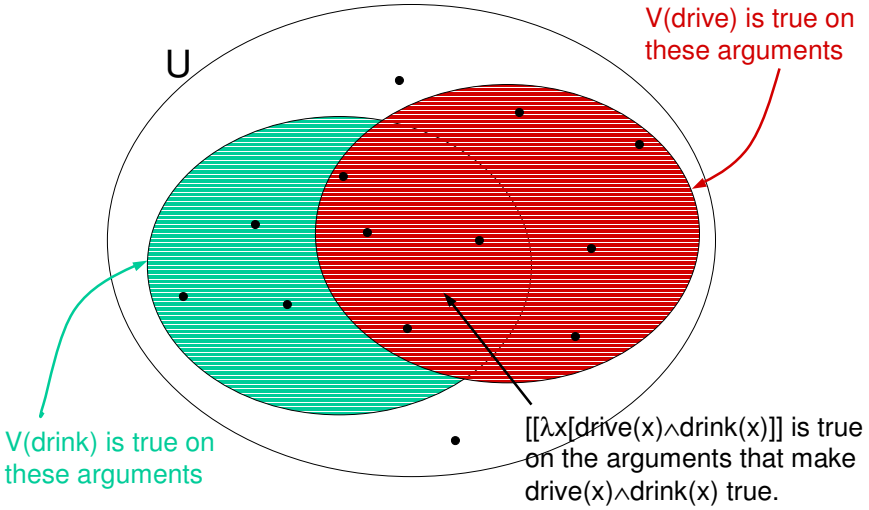
λx[drive(x)∧drink(x)]: <e,t>

Coverage problem: Solved!

Swimming is healthy
Not smoking is healthy
Driving and drinking is dangerous

John drives and drinks

Semantics of λ-expressions



Semantics of λ-expressions

- If $\alpha \in WE_\tau$, $v \in Var_\sigma$, then $[[\lambda v \alpha]]^{M,g}$ is that function $f : D_\sigma \rightarrow D_\tau$ such that for all $a \in D_\sigma$, $f(a) = [[\alpha]]^{M,g[v/a]}$
- Notice that of course $f \in D_{<\sigma,\tau>}$.
- In general: $[[(\lambda v \alpha)(\beta)]]^{M,g} = [[\alpha]]^{M,g[v/\ [[\beta]]^{M,g}]}$

A syntactic shortcut for the evaluation of λ-expressions

- By the modified variable assignment, the value of the argument of the λ-expression is passed through its body and becomes the value of all occurrences of variables bound by the λ-operator.
- We obtain the same result, if we first substitute the free occurrences of the λ-variable in $\lambda v \alpha(\beta)$ by the argument β, and only then interpret the result:
  - $[[\lambda v \alpha(\beta)]]^{M,g} = [[\alpha]]^{M,g[v/\ [[\beta]]^{M,g}]}$ to
  - $[[\lambda v \alpha(\beta)]]^{M,g} = [[\ [\beta/v]\alpha\ ]]^{M,g}$
- This is the basic idea behind the λ-calculus.

Variable capturing

- Are $\lambda v\alpha(\beta)$ and $[\beta/v]\alpha$ always equivalent?
  - $\lambda x[\text{drive}(x) \wedge \text{drink}(x)](\text{john}) \Rightarrow \text{drive}(\text{john}) \wedge \text{drink}(\text{john})$
  - $\lambda x[\text{drive}(x) \wedge \text{drink}(x)](y) \Rightarrow \text{drive}(y) \wedge \text{drink}(y)$
  - $\lambda x\, [\forall y\, \text{know}(x)(y)]\, (\text{john}) \Rightarrow \forall y\, \text{know}(\text{john})(y)$
  - $\lambda x\, [\forall y\, \text{know}(x)(y)]\, (y) \not\Rightarrow \forall y\, \text{know}(y)(y)$
- Let v, v' be variables of the same type, α 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 in the λ-calculus

- β-conversion:

  $\lambda v\alpha(\beta) \Leftrightarrow {}^{[\beta/v]}\alpha$, if all free variables in β are free for v in α.
- α-conversion:

  $\lambda v\alpha \Leftrightarrow \lambda v'\, {}^{[v'/v]}\alpha$, if v' is free for v in α.
- η-conversion:

  $\lambda v(\alpha(v)) \Leftrightarrow \alpha$

The rule which we will use most in semantics construction is β-conversion in the left-to-right direction (β-reduction), which allows us to simplify representations.

Example

John drives and drinks.

drive: <e,t> x:e drink: <e,t> x:e

drive(x): t drink(x): t

drive(x)∧drink(x): t

john : e λx[drive(x)∧drink(x)]: <e,t>

(λx[drive(x)∧drink(x)])(john) : t

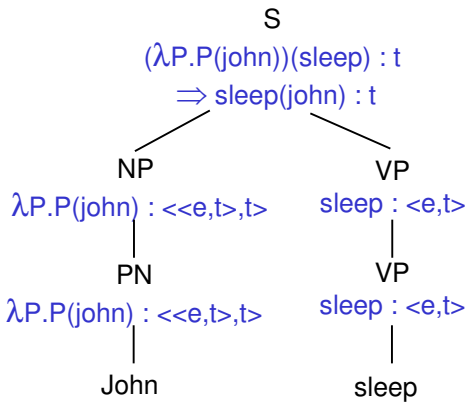
$\Rightarrow_\beta$ drive(john) ∧ drink(john) : t

Back to noun phrases

- We were looking for a uniform representation for NPs.
  - proper names: could use constants of type e
  - quantified NPs: ???
- Solve this problem using lambdas for type raising:
  - All NPs are represented as terms of type <<e,t>,t>.
  - Interpretation of "John": The property P (of type <e,t>) belongs to this set if John has it.
  - Interpretation of "every student": P belongs to the set if every student has it.
  - and so on

Semantics construction

- John sleeps.

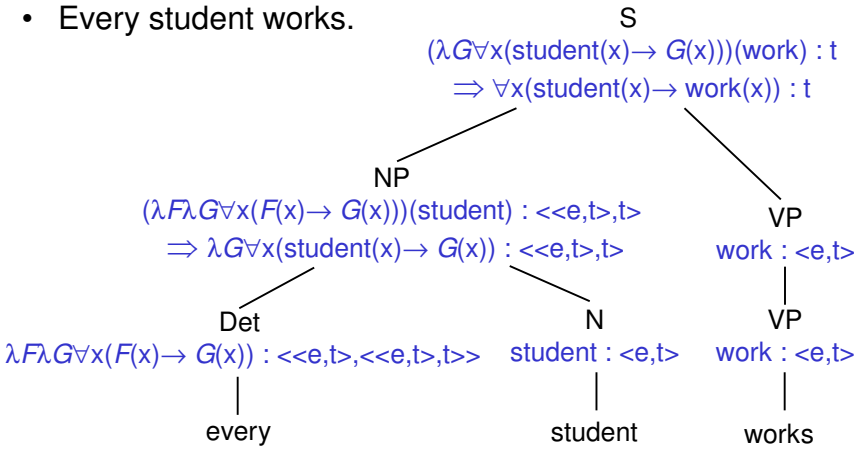


Semantic representations for NPs

- *every student* denotes a second-order property (<<e,t>,t>) which holds of a (first-oder) property ω iff all students are in ω.
- This semantic information can be straightforwardly encoded as a lambda term:
  $\lambda G\, \forall x(\text{student}(x) \rightarrow G(x))$
- Accordingly, the determiner *every* can be represented as:
  $\lambda F \lambda G \forall x(F(x) \rightarrow G(x))$

Semantics construction

- Every student works.



More noun phrases

John	$\lambda G[G(j^*)]$
Somebody	$\lambda G\, \exists x G(x)$
A student	$\lambda G\, \exists x(\text{student}(x) \wedge G(x))$
No student	$\lambda G \neg \exists x\, (\text{student}(x) \wedge G(x))$
John	$\lambda G[G(j^*)]$
John and Mary	$\lambda G[G(j^*) \wedge G(m^*)]$

More determiners

a, some	$\lambda F \lambda G\, \exists x(F(x) \wedge G(x))$
no	$\lambda F \lambda G\, \neg \exists x\, (F(x) \wedge G(x))$
most	most (a constant)

Conclusion

- We wanted compositional semantics construction.
  - This turned out to be not so easy for nontrivial sentences.
  - With lambdas, it is easy!
  - Breaking through some expressive limits of lambda-free type theory.
- Lambda abstraction is a very natural and straightforward extension to lambda-free type theory, and belongs to standard definitions of type theory.