Semantic Theory

Semantics Construction

Manfred Pinkal
Stefan Thater

Summer 2007

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

The Principle of Compositionality

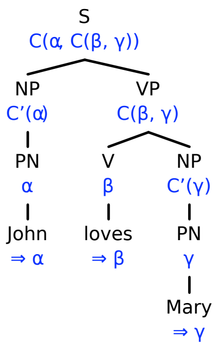
- The meaning of a complex expression is uniquely determined by the meanings of its sub-expressions and the syntactic rules by which they are combined.
- (The principle is also called “Frege’s principle”)

Two Levels of Interpretation

- Semantic interpretation is a two-step process
  - Natural language (NL) expressions are assigned a semantic representation (logical formulas).
  - 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.

Compositional Semantics Construction

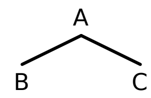
- Basic idea: we start with a syntactic analysis of an NL expression, and
- assign each syntactic node in the syntax tree a semantic representation
- by combining the representations of its daughter nodes.



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 for non-branching nodes

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

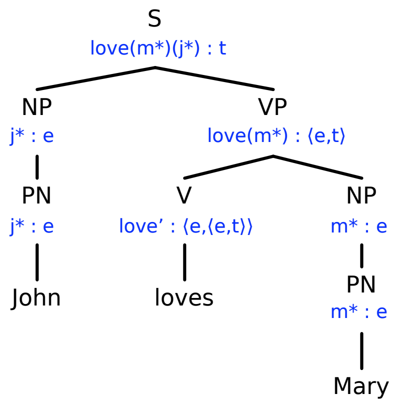
Basic Rules

- Rule of functional application

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

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

An Example



Noun phrases and compositionality

Every student presents a paper

$\forall x(\text{student}'(x) \rightarrow \exists y(\text{paper}'(y) \wedge \text{present}'(y)(x))$

Noun phrases and compositionality

Every student presents a paper

$\forall x(\text{student}'(x) \rightarrow \exists y(\text{book}'(y) \wedge \text{present}'(y)(x))$

Noun phrases and compositionality

"John works" ⇒ work'(j*)

"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'(j*) ∧ work(m*)

- What's the semantic representation of a noun phrase?

Towards a unified semantics of NPs

- "John works."

$$\frac{\text{j*} : e \qquad \text{work}' : \langle e,t\rangle}{\text{work}'(\text{j*}) : t}$$

- "Every student works."

$$\frac{\text{every-student}' : \langle\langle e,t\rangle,t\rangle \qquad \text{work}' : \langle e,t\rangle}{\text{every-student}'(\text{work}') : t}$$

Towards a unified semantics of NPs

- "John works."

$$\frac{\text{john'} : \langle\langle e,t\rangle,t\rangle \quad \text{work'} : \langle e,t\rangle}{\text{john'(work')} : t}$$

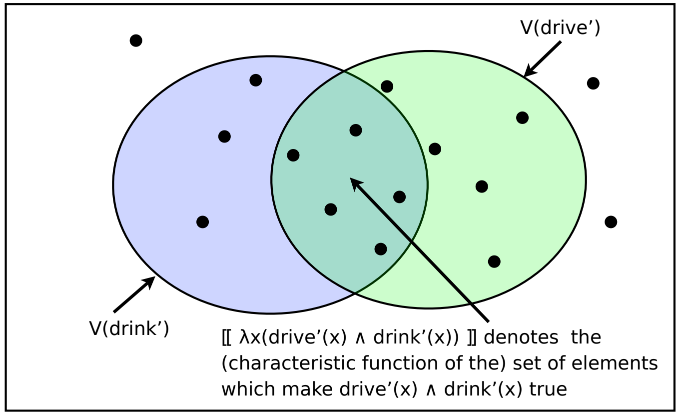
- "Every student works."

$$\frac{\text{every-student'} : \langle\langle e,t\rangle,t\rangle \quad \text{work'} : \langle e,t\rangle}{\text{every-student'(work')} : t}$$

λ-Abstraction

- λx(drive(x) ∧ 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) ∧ drink(x)' results in the complex predicate 'λx(drive(x) ∧ drink(x)).'

λx(drive'(x) ∧ drink'(x))



λ-Abstraction: Semantics

- 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 $f \in D_{\langle\sigma,\tau\rangle}$.
- In general: $[\![(\lambda v\alpha)(\beta)]\!]^{M,g} = [\![\alpha]\!]^{M,g[v / [\![\beta]\!]M,g]}$

β-Reduction

- 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?
  - $\lambda x[\text{drive}'(x) \wedge \text{drink}'(x)](j^*) \Rightarrow \text{drive}'(j^*) \wedge \text{drink}'(j^*)$
  - $\lambda x[\text{drive}'(x) \wedge \text{drink}'(x)](y) \Rightarrow \text{drive}'(y) \wedge \text{drink}'(y)$
  - $\lambda x[\forall y\ \text{know}'(x)(y)](j^*) \Rightarrow \forall y\ \text{know}(j^*)(y)$
  - $\lambda x[\forall y\ \text{know}'(x)(y)](y) \not\Rightarrow \forall y\ \text{know}(y)(y)$
- 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.

An 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

λx (drive'(x) ∧ drink'(x)) : ⟨e,t⟩ j* : e

λx (drive'(x) ∧ drink'(x)) (j*)

$\Rightarrow_\beta$ drive'(j*) ∧ drink'(j*)

Back to Noun Phrases

- We were looking for a uniform representation for noun phrases:
  - All noun phrases are uniformly represented as terms of type ⟨⟨e,t⟩,t⟩ i.e., expressions that denote sets of first-order properties P (type ⟨e,t⟩).
  - Interpretation of "John:" the set of properties P such that John has property P.
  - Interpretation of "every student:" the set of properties P such that every student has P.
  - and so on ...

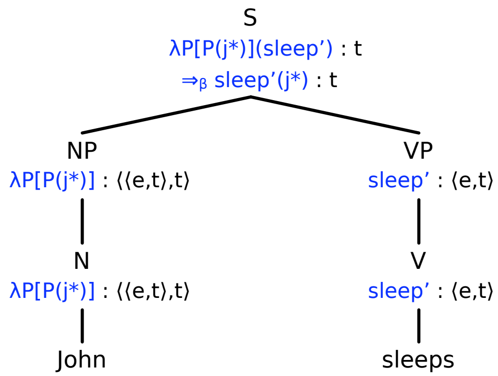
Back to Noun Phrases

- Interpretation of "John:" the set of properties P such that John has property P:
  - $\lambda P[P(j^*)]$
- Interpretation of "every student:" P belongs to the set if every student has property P:
  - $\lambda P[\forall x(\text{student}'(x) \rightarrow P(x))]$
- Interpretation of "a student:" P belongs to the set if a student has property P:
  - $\lambda P[\exists x(\text{student}'(x) \wedge P(x))]$

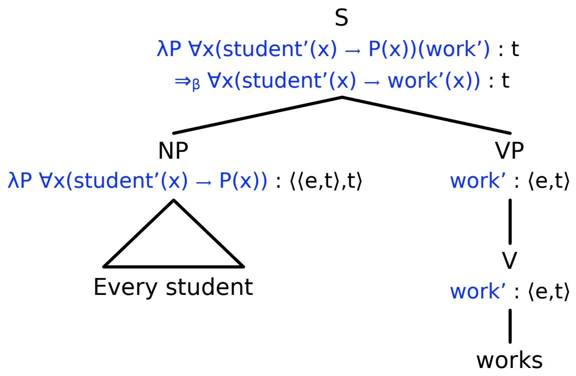
More Noun Phrases

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

"John sleeps"



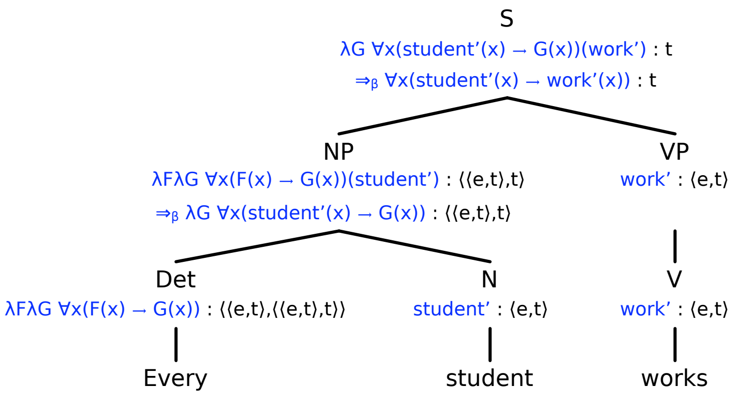
“Every student works”



Determiners

a, some	$\Rightarrow \lambda F \lambda G\ \exists x(F(x) \wedge G(x))$
every	$\Rightarrow \lambda F \lambda G\ \forall x(F(x) \rightarrow G(x))$
no	$\Rightarrow \lambda F \lambda G\ \neg\exists x(F(x) \wedge G(x))$
most	$\Rightarrow$ most’ (a constant)

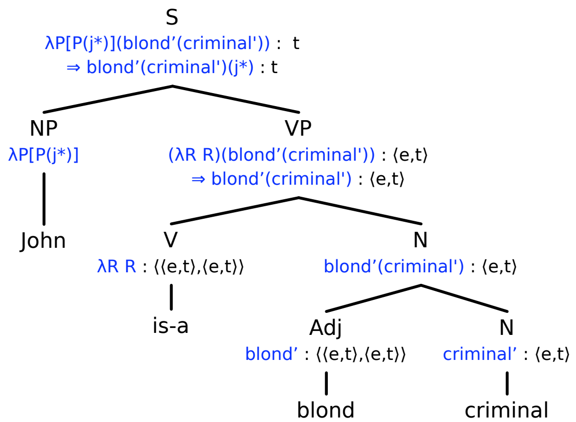
“Every student works.”



Back to Adjectives

- “John is a blond criminal”
  - criminal’(j*) ∧ blond’(j*)
- “John is a famous criminal”
  - criminal’(j*) ∧ famous’(j*) ?
- “John is an alleged criminal”
  - criminal’(j*) ∧ alleged’(j*) ???

"John is a blond criminal."



Adjective Classes

- Adjectives can be classified with respect to the way they combine the adjective and noun meanings:
  - intersective adjectives (blond, carnivorous, ...): $[\![\text{blond N}]\!] = [\![\text{blond}]\!] \cap [\![\text{N}]\!]$
  - subsective adjectives (skillful, typical, ...): $[\![\text{skillful N}]\!] \subseteq [\![\text{N}]\!]$
  - privative adjectives (past, fake, ...): $[\![\text{past N}]\!] \cap [\![\text{N}]\!] = \emptyset$
  - there are also other non-subsective adjectives that are not privative (alleged, ...)

A new problem with adjectives

- We want the best of both worlds:
  - compositional semantics construction
  - explicit and meaningful final semantic representations
- We don't have this yet for intersective adjectives.
- We can get this in two different ways
  - use meaning postulates
  - use more explicit lambda terms

Meaning Postulates

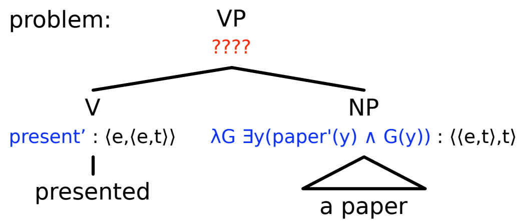
- Characterise the meaning of a predicate that stands for a word (e.g., "blond") by using logical axioms.
- Meaning postulate for intersective adjectives:
  - $\forall P \forall x(\text{blond}'(P)(x) \rightarrow P(x))$
- These axioms would be part of our background knowledge.
- For example, we could infer "criminal(john)" from "blond(criminal)(john)" and this axiom.

More explicit lambda terms

- For intersective adjectives, we can also do it by assigning the word a more elaborate lambda term:
  - λPλx(P(x) ∧ blond'(P)(x))
- or alternatively as
  - λPλx(P(x) ∧ blond*(x))
  - where "blond*" is a constant of type ⟨e,t⟩ which should denote the set of blond individuals in the universe.
- This will beta-reduce to the formula we want.

Transitive Verbs

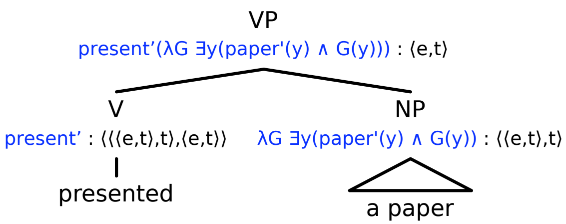
- A composition problem:



every student ⇒ λF ∀x(student'(x) → F(x)) : ⟨⟨e,t⟩,t⟩
a paper ⇒ λG ∃y(paper(y) ∧ G(y)) : ⟨⟨e,t⟩,t⟩
presented ⇒ present: ⟨e,⟨e,t⟩⟩

The solution: Type-Raising

- Raise the type of the first-order relation:
  - present: ⟨⟨⟨e,t⟩,t⟩,⟨e,t⟩⟩



Transitive Verbs

- But now our semantic representation no longer beta-reduces to a FOL formula:
  - ∀x[student(x) → present(λG∃y paper(y) ∧G(y))(x)]
- Same problem as with intersective adjectives, same solution.
- Represent transitive verbs like "present" as follows:
  - λQλx[Q(λy[present*(y)(x)])]: ⟨⟨⟨e,t⟩,t⟩,⟨e,t⟩⟩,
  - where present*: ⟨e,⟨e,t⟩⟩

“… presented a paper”

- a paper ⇒ λG∃z(paper’(z) ∧ G(z))
- presented ⇒ λQλx[Q(λy[present*(y)(x)])]
- presented a paper
  - ⇒ λQλx[Q(λy[present*(y)(x)])](λG∃z(paper’(z) ∧ G(z)))
  - ⇒ λx[λG∃z(paper’(z) ∧ G(z))(λy[present*(y)(x)])]
  - ⇒ λx[∃z(paper’(z) ∧ λy[present*(y)(x)](z))]
  - ⇒ λx[∃z(paper’(z) ∧ present*(z)(x))]

Conclusion

- Semantics construction is not so easy for nontrivial sentences.
- With lambda-abstraction and application, these sentences can be treated in a straightforward way.
- Lambda abstraction is a very natural and straightforward extension to lambda-free type theory, and belongs to standard definitions of type theory.