Syntax/Semantics interface (Semantic analysis)

Sharon Goldwater
(based on slides by James Martin and Johanna Moore)

15 November 2019

School of informatics

Last time

- Discussed properties we want from a meaning representation:
  - compositional
  - verifiable
  - canonical form
  - unambiguous
  - expressive
  - allowing inference
- Argued that first-order logic has all of these except compositionality, and is a good fit for natural language.
- Adding λ-expressions to FOL allows us to compute meaning representations compositionally.

Today

- We'll see how to use λ-expressions in computing meanings for sentences: syntax-driven semantic analysis.
- But first: a final improvement to event representations

Verbal (event) MRs: the story so far

Syntax:

NP give NP_1 NP_2

Semantics:

$\lambda z.\ \lambda y.\ \lambda x.\ \mathit{Giving}_1(x,y,z)$

Applied to arguments:

$\lambda z.\ \lambda y.\ \lambda x.\ \mathit{Giving}_1(x,y,z)\ \mathit{(book)(Mary)(John)}$

As in the sentence:

John gave Mary a book.

$\mathit{Giving}_1\mathit{(John, Mary, book)}$

But what about these?

John gave Mary a book for Susan.
Giving$_2$(John, Mary, Book, Susan)

John gave Mary a book for Susan on Wednesday.
Giving$_3$(John, Mary, Book, Susan, Wednesday)

John gave Mary a book for Susan on Wednesday in class.
Giving$_4$(John, Mary, Book, Susan, Wednesday, InClass)

John gave Mary a book with trepidation.
Giving$_5$(John, Mary, Book, Susan, Trepidation)

Problem with event representations

- Predicates in First-order Logic have fixed arity
- Requires separate *Giving* predicate for each syntactic **subcategorisation frame** (number/type/position of arguments).
- Separate predicates have no logical relation, but they ought to.
  - Ex. if *Giving$_3$(a, b, c, d, e)* is true, then so are *Giving$_2$(a, b, c, d)* and *Giving$_1$(a, b, c)*.
- See J&M for various unsuccessful ways to solve this problem; we'll go straight to a more useful way.

Reification of events

- We can solve these problems by **reifying** events.
  - Reify: to "make real" or concrete, i.e., give events the same status as entities.
  - In practice, introduce variables for events, which we can quantify over.

Reification of events

- We can solve these problems by **reifying** events.
  - Reify: to "make real" or concrete, i.e., give events the same status as entities.
  - In practice, introduce variables for events, which we can quantify over.
- MR for John gave Mary a book is now

$$\exists\ e, z.\ \mathit{Giving}(e) \wedge \mathit{Giver}(e, \mathit{John}) \wedge \mathit{Givee}(e, \mathit{Mary}) \wedge \mathit{Given}(e,z) \wedge \mathit{Book}(z)$$

- The giving event is now a single predicate of arity 1: *Giving(e)*; remaining conjuncts represent the participants (semantic roles).

Entailment relations

- This representation automatically gives us logical **entailment** relations between events. ("A **entails** B" means "A ⇒ B".)
- John gave Mary a book on Tuesday entails
  John gave Mary a book.

Entailment relations

- This representation automatically gives us logical **entailment** relations between events. ("A **entails** B" means "A ⇒ B".)
- John gave Mary a book on Tuesday entails
  John gave Mary a book. Similarly,

$$\exists\, e, z.\ \mathit{Giving}(e) \wedge \mathit{Giver}(e, \mathit{John}) \wedge \mathit{Givee}(e, \mathit{Mary}) \wedge \mathit{Given}(e,z) \wedge \mathit{Book}(z) \wedge \mathit{Time}(e, \mathit{Tuesday})$$

entails

$$\exists\, e, z.\ \mathit{Giving}(e) \wedge \mathit{Giver}(e, \mathit{John}) \wedge \mathit{Givee}(e, \mathit{Mary}) \wedge \mathit{Given}(e,z) \wedge \mathit{Book}(z)$$

- Can add as many semantic roles as needed for the event.

At last: Semantic Analysis

- Given this way of representing meanings, how do we compute meaning representations from sentences?
- The task of **semantic analysis** or **semantic parsing**.
- Most methods rely on a (prior or concurrent) syntactic parse.
- Here: a compositional **rule-to-rule** approach based on FOL augmented with λ-expressions.

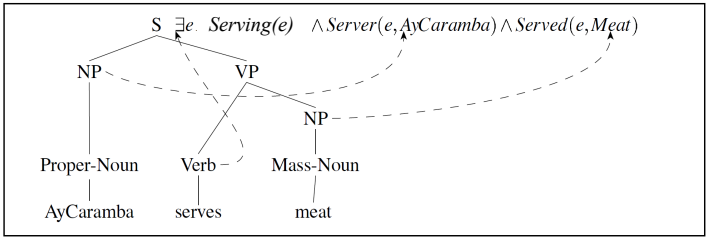
Syntax Driven Semantic Analysis

- Based on the **principle of compositionality**.
  - meaning of the whole built up from the meaning of the parts
  - more specifically, in a way that is guided by word order and syntactic relations.
- Build up the MR by augmenting CFG rules with semantic composition rules.
- Representation produced is *literal meaning*: context independent and free of inference

Note: other syntax-driven semantic parsing formalisms exist, e.g. Combinatory Categorial Grammar (Steedman, 2000) has seen a surge in popularity recently.

Example of final analysis

- What we're hoping to build



CFG Rules with Semantic Attachments

- To compute the final MR, we add **semantic attachments** to our CFG rules.
- These specify how to compute the MR of the parent from those of its children.
- Rules will look like:

$$A \rightarrow \alpha_1 \ldots \alpha_n \qquad \{f(\alpha_j.sem, \ldots, \alpha_k.sem)\}$$

- $A.sem$ (the MR for A) is computed by applying the function f to the MRs of some subset of A's children.

Proposed rules

- Ex: AyCaramba serves meat (with parse tree)
- Rules with semantic attachments for nouns and NPs:

ProperNoun → AyCaramba	{*AyCaramba*}
MassNoun → meat	{*Meat*}
NP → ProperNoun	{*ProperNoun.sem*}
NP → MassNoun	{*MassNoun.sem*}

- Unary rules normally just copy the semantics of the child to the parents (as in NP rules here).

What about verbs?

- Before event reification, we had verbs with meanings like:

λy. λx. Serving(x,y)

- λs allowed us to compose arguments with predicate.
- We can do the same with reified events:

λy. λx. ∃e. Serving(e) ∧ Server(e, x) ∧ Served(e, y)

What about verbs?

- Before event reification, we had verbs with meanings like:

$$\lambda y.\ \lambda x.\ Serving(x,y)$$

- λs allowed us to compose arguments with predicate.
- We can do the same with reified events:

$$\lambda y.\ \lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, y)$$

- This MR is the semantic attachment of the verb:

Verb $\rightarrow$ serves
{ $\lambda y.\ \lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, y)$ }

Building larger constituents

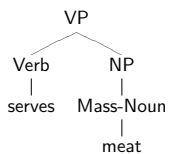
- The remaining rules specify how to apply λ-expressions to their arguments. So, VP rule is:

VP $\rightarrow$ Verb NP {*Verb.sem(NP.sem)*}

Building larger constituents

- The remaining rules specify how to apply λ-expressions to their arguments. So, VP rule is:

VP $\rightarrow$ Verb NP {*Verb.sem(NP.sem)*}



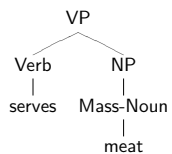
where *Verb.sem* = $\lambda y.\ \lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, y)$

and *NP.sem* = *Meat*

Building larger constituents

- The remaining rules specify how to apply λ-expressions to their arguments. So, VP rule is:

VP $\rightarrow$ Verb NP {*Verb.sem(NP.sem)*}



where *Verb.sem* = $\lambda y.\ \lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, y)$

and *NP.sem* = *Meat*

- So, *VP.sem* = $\lambda y.\ \lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, y)$ *(Meat)* = $\lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, Meat)$

Finishing the analysis

- Final rule is:

  S → NP VP {*VP.sem(NP.sem)*}

- now with *VP.sem* =
  $\lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, Meat)$

  and *NP.sem* =
  AyCaramba

- So, *S.sem* =
  $\lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, Meat)$ *(AyCa.)* =
  $\exists e.\ Serving(e) \wedge Server(e, AyCaramba) \wedge Served(e, Meat)$

Problem with these rules

- Consider the sentence Every child sleeps.

  $$\forall x.\ Child(x) \Rightarrow \exists e.\ Sleeping(e) \wedge Sleeper(e, x)$$

- Meaning of Every child (involving x) is interleaved with meaning of sleeps

- As next slides show, our existing rules can't handle this example, or quantifiers (from NPs with determiners) in general.

- We'll show the problem, then the solution.

Breaking it down

- What is the meaning of Every child anyway?

- Every child ...

...sleeps	$\forall x.\ Child(x) \Rightarrow \exists e.\ Sleeping(e) \wedge Sleeper(e, x)$
...cries	$\forall x.\ Child(x) \Rightarrow \exists e.\ Crying(e) \wedge Crier(e, x)$
...talks	$\forall x.\ Child(x) \Rightarrow \exists e.\ Talking(e) \wedge Talker\ (e, x)$
...likes pizza	$\forall x.\ Child(x) \Rightarrow \exists e.\ Liking\ (e) \wedge Liker(e, x) \wedge Likee(e, pizza)$

Breaking it down

- What is the meaning of Every child anyway?

- Every child ...

...sleeps	$\forall x.\ Child(x) \Rightarrow \exists e.\ Sleeping(e) \wedge Sleeper(e, x)$
...cries	$\forall x.\ Child(x) \Rightarrow \exists e.\ Crying(e) \wedge Crier(e, x)$
...talks	$\forall x.\ Child(x) \Rightarrow \exists e.\ Talking(e) \wedge Talker\ (e, x)$
...likes pizza	$\forall x.\ Child(x) \Rightarrow \exists e.\ Liking\ (e) \wedge Liker(e, x) \wedge Likee(e, pizza)$

- So it looks like the meaning is something like

  $$\forall x.\ Child(x) \Rightarrow Q(x)$$

- where $Q(x)$ is some (potentially quite complex) expression with a predicate-like meaning

Could this work with our rules?

- We said *S.sem* should be *VP.sem(NP.sem)*
- but

  $\lambda y.\ \exists\ e.\ Sleeping(e) \wedge Sleeper(e, y)\ (\forall\ x.\ Child(x) \Rightarrow Q(x))$

  yields

  $\exists\ e.\ Sleeping(e) \wedge Sleeper(e, \forall\ x.\ Child(x) \Rightarrow Q(x))$
- This isn't a valid FOL: complex expressions cannot be arguments to predicates.

Switching things around

- But if we define *S.sem* as *NP.sem(VP.sem)* it works!
- First, must make *NP.sem* into a functor by adding λ:

  $\lambda Q\ \forall\ x.\ Child(x) \Rightarrow Q(x)$

Switching things around

- But if we define *S.sem* as *NP.sem(VP.sem)* it works!
- First, must make *NP.sem* into a functor by adding λ:

  $\lambda Q\ \forall\ x.\ Child(x) \Rightarrow Q(x)$
- Then, apply it to *VP.sem*:

$\lambda Q\ \forall\ x.\ Child(x) \Rightarrow Q(x)\ (\lambda y.\ \exists\ e.\ Sleeping(e) \wedge Sleeper(e, y))$

$\forall\ x.\ Child(x) \Rightarrow (\lambda y.\ \exists\ e.\ Sleeping(e) \wedge Sleeper(e, y))\ (x)$

$\forall\ x.\ Child(x) \Rightarrow \exists\ e.\ Sleeping(e) \wedge Sleeper(e, x)$

But, how can we get the right NP.sem?

- We will need a new set of noun rules:

Noun $\rightarrow$ Child	$\{\lambda x.\ Child(x)\}$
Det $\rightarrow$ Every	$\{\lambda P.\ \lambda Q.\ \forall\ x.\ P(x) \Rightarrow Q(x)\}$
NP $\rightarrow$ Det Noun	$\{Det.sem(Noun.sem)\}$

But, how can we get our NP.sem?

- We will need a new set of noun rules:

  Noun → Child {$\lambda x.$ *Child(x)*}
  Det → Every {$\lambda P.\ \lambda Q.\ \forall\ x.$ *P(x)* $\Rightarrow$ *Q(x)*}
  NP → Det Noun {*Det.sem(Noun.sem)*}

- So, Every child is derived as

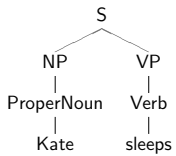
  $\lambda P.\ \lambda Q.\ \forall\ x.$ *P(x)* $\Rightarrow$ *Q(x)* *(*$\lambda x.$ *Child(x))*

  $\lambda Q\ \forall\ x.$ *(*$\lambda x.$ *Child(x))*(x) $\Rightarrow$ *Q(x)*

  $\lambda Q\ \forall\ x.$ *Child(x)* $\Rightarrow$ *Q(x)*

One last problem

- Our previous MRs for proper nouns were not functors, so don't work with our new rule S → NP VP {*NP.sem(VP.sem)*}.



Kate *(*$\lambda y.\ \exists\ e.$ *Sleeping(e)* $\wedge$ *Sleeper(e, y))*

$\Rightarrow$ Not valid!

λ to the rescue again

- Assign a different MR to proper nouns, allowing them to take VPs as arguments:

  ProperNoun → Kate {$\lambda P.$ *P(Kate)*}

- For Kate sleeps, this gives us

  $\lambda P.$ *P(Kate)* *(*$\lambda y.\ \exists\ e.$ *Sleeping(e)* $\wedge$ *Sleeper(e, y))*
  ($\lambda y.\ \exists\ e.$ *Sleeping(e)* $\wedge$ *Sleeper(e, y))(Kate)*
  $\exists\ e.$ *Sleeping(e)* $\wedge$ *Sleeper(e, Kate))*

λ to the rescue again

- Assign a different MR to proper nouns, allowing them to take VPs as arguments:

  ProperNoun → Kate {$\lambda P.$ *P(Kate)*}

- For Kate sleeps, this gives us

  $\lambda P.$ *P(Kate)* *(*$\lambda y.\ \exists\ e.$ *Sleeping(e)* $\wedge$ *Sleeper(e, y))*
  ($\lambda y.\ \exists\ e.$ *Sleeping(e)* $\wedge$ *Sleeper(e, y))(Kate)*
  $\exists\ e.$ *Sleeping(e)* $\wedge$ *Sleeper(e, Kate))*

- Terminology: we **type-raised** the the argument a of a function f, turning it into a function g that takes f as argument. (!)
  - The final returned value is the same in either case.

Final grammar?

S → NP VP	{*NP.sem(VP.sem)*}
VP → Verb	{*Verb.sem*}
VP → Verb NP	{*Verb.sem(NP.sem)*}
NP → Det Noun	{*Det.sem(Noun.sem)*}
NP → ProperNoun	{*ProperNoun.sem*}
Det → Every	{$\lambda P.\ \lambda Q.\ \forall x.\ P(x) \Rightarrow Q(x)$}
Noun → Child	{$\lambda x.\ Child(x)$}
ProperNoun → Kate	{$\lambda P.\ P(Kate)$}
Verb → sleeps	{$\lambda x.\ \exists e.\ Sleeping(e) \wedge Sleeper(e, x)$}
Verb → serves	{$\lambda y.\ \lambda x.\ \exists e.\ Serving(e) \wedge Server(e, x) \wedge Served(e, y)$}

Complications

- This grammar still applies Verbs to NPs when *inside* the VP.
- Try doing this with our new type-raised NPs and you will see it doesn't work.
- In practice, we need automatic type-raising rules that can be used exactly when needed, otherwise we keep the base type.
  - e.g., "base type" of proper noun is "entity", not "function from (functions from entities to truth values) to truth values".

What we did achieve

Developed a grammar with semantic attachments using many ideas now in use:

- existentially quantified variables represent events
- lexical items have function-like λ-expressions as MRs
- non-branching rules copy semantics from child to parent
- branching rules apply semantics of one child to the other(s) using λ-reduction.

Semantic parsing algorithms

- Given a CFG with semantic attachments, how do we obtain the semantic analysis of a sentence?
- One option (integrated): Modify syntactic parser to apply semantic attachments at the time syntactic constituents are constructed.
- Second option (pipelined): Complete the syntactic parse, then walk the tree bottom-up to apply semantic attachments.

Learning a semantic parser

- Much current research focuses on *learning* semantic grammars rather than *hand-engineering* them.
- Given sentences paired with meaning representations, e.g.,

Every child sleeps	$\forall x.\ Child(x) \Rightarrow \exists e.\ Sleeping(e) \wedge Sleeper(e, x)$
AyCaramba serves meat	$\exists e.\ Serving(e) \wedge Server(e, AyCaramba) \wedge Served(e, Meat)$

- Can we automatically learn
  - Which words are associated with which bits of MR?
  - How those bits combine (in parallel with the syntax) to yield the final MR?
- And, can we do this with less well-specified semantic representations?

See, e.g., **????**

Summary

- Semantic analysis/semantic parsing: the process of deriving a meaning representation from a sentence.
- Uses the grammar and lexicon (augmented with semantic information) to create context-independent literal meanings
- λ-expressions handle compositionality, building semantics of larger forms from smaller ones.
- Final meaning representations are expressions in first-order logic.