An Introduction to Formal Computational Semantics

CS224N/Ling 280

Christopher Manning

May 23, 2000; revised 2008

A first example

- \[ S : \text{respect}(\text{kathy}, \text{fong}) \]
- \[ \text{NP} : \text{kathy} \]
- \[ \text{VP} : \lambda x.\text{respect}(x, \text{fong}) \]
- \[ \text{Kathy} \]
- \[ \text{Fong} \]
- \[ \text{respects} \]
- \[ \text{runs} \]

- \[ \{ \text{VP respects Fong} \} : [\lambda y.\lambda x.\text{respect}(x, y)](\text{fong}) \]
  \[ = \lambda x.\text{respect}(x, \text{fong}) \]
  \[ [\text{5 Kathy respects Fong}] : [\lambda x.\text{respect}(x, \text{fong})](\text{kathy}) \]
  \[ = \text{respect}(\text{kathy}, \text{fong}) \]

Database/knowledgebase interfaces

- Assume that \text{respect} is a table \text{Respect} with two fields \text{respecter} and \text{respected}
- Assume that \text{kathy} and \text{fong} are IDs in the database: \text{k} and \text{f}
- If we assert \text{Kathy respects Fong} we might evaluate the form \[ \text{respect}(\text{fong})(\text{kathy}) \] by doing an insert operation:
  insert into \text{Respects}(\text{respecter}, \text{respected}) values (\text{k}, \text{f})

Typed \( \lambda \) calculus (Church 1940)

- Everything has a type (like Java!)
- \text{Bool} true values (0 and 1)
- \text{Ind} individuals
- \text{Ind} \rightarrow \text{Bool} properties
- \text{Ind} \rightarrow \text{Ind} \rightarrow \text{Bool} binary relations
- \text{kathy} and \text{fong} are \text{Ind}
- \text{run} is \text{Ind} \rightarrow \text{Bool}
- \text{respect} is \text{Ind} \rightarrow \text{Ind} \rightarrow \text{Bool}
- Types are interpreted right associatively.
- \text{respect} is \text{Ind} \rightarrow (\text{Ind} \rightarrow \text{Bool})
- We convert a several argument function into embedded unary functions. Referred to as currying.
Typed λ calculus (Church 1940)

- Once we have types, we don’t need λ variables just to show what arguments something takes, and so we can introduce another operation of the λ calculus:
  - η reduction [abstractions can be contracted]
    \[ \lambda x.(P(x)) \to P \]
  - This means that instead of writing:
    \[ \lambda y.\lambda x.\text{respect}(x,y) \]
    we can just write:
    \[ \text{respect} \]

Types of major syntactic categories

- nouns and verb phrases will be properties (Ind → Bool)
- noun phrases are Ind - though they are commonly type-raised to (Ind → Bool) → Bool
- adjectives are (Ind → Bool) → (Ind → Bool)
  This is because adjectives modify noun meanings, that is properties.
- Intensifiers modify adjectives: e.g, very in a very happy camper, so they’re (((Ind → Bool) → (Ind → Bool)) → ((Ind → Bool) → (Ind → Bool))) [honest!].

Types of major syntactic categories

A grammar fragment

- Kathy, NP : kathyInd
  Fong, NP : fongInd
  Palo Alto, NP : paloaltoInd
  car, N : carInd → Bool
  overpriced, Adj : overpricedInd → (Ind → Bool)
  outside, PP : outsideInd → (Ind → Bool)
  red, Adj : λP.(λx.P(x) ∧ red′(x))
  in, P : λy.λP.λx.(P(x) ∧ in′(y)(x))
  the, Det : t
  a, Det : someInd → (Ind → Bool)
  respects, V : respectInd → (Ind → Bool)
  runs, V : runInd → (Ind → Bool)
  likes, V : likeInd → (Ind → Bool)
A grammar fragment

- in′ is Ind → Ind → Bool
- in ≡ λy.λP.λx.(P(x) ∧ in′(y)(x)) is Ind → (Ind → Bool) → (Ind → Bool)
- red′ is Ind → Bool
- red ≡ λP.λx.(P(x) ∧ red′(x)) is (Ind → Bool) → (Ind → Bool)

Curried multi-argument functions

Quiz question

- Which individuals are the red things in Palo Alto?
- Who respects kathy (k)?

Adjective and PP modification

Intersective adjectives

- Syntactic ambiguity is spurious: you get the same semantics either way
- Database evaluation is possible via a table join

Non-intersective adjectives

- For non-intersective adjectives get different semantics depending on what they modify
- overpriced(in(paloalto)(house))
- in(paloalto)(overpriced(house))
- But probably won’t be able to evaluate it on database!
**Adding more complex NPs**

NP: A man $\rightarrow \exists x.\text{man}(x)$  
S: A man loves Mary  
$\rightarrow \ast \text{love}(\exists x.\text{man}(x), \text{mary})$

- How to fix this?

**A solution for quantifiers**

What we want is:

"A man loves Mary" $\rightarrow \exists z.\text{man}(z) \land \text{love}(z, \text{mary})$

What we have is:

"man" $\rightarrow \lambda y.\text{man}(y)$
"loves Mary" $\rightarrow \lambda x.\text{love}(x, \text{mary})$

How about: $\exists z. (\lambda y.\text{man}(y)(z) \land \lambda x.\text{love}(x, \text{mary})(z))$

Remember: We can use variables for any kind of term.
So next:

$\lambda P(\lambda Q.\exists z (P(z) \land Q(z))) \iff \text{"A"}$

**A disappointment**

Our first idea for NPs with determiner didn’t work out:

"A man" $\rightarrow \exists z.\text{man}(z)$
"A man loves Mary" $\rightarrow \ast \text{love}(\exists z.\text{man}(z), \text{mary})$

But what was the idea after all?

Nothing!

$\exists z.\text{man}(z)$ just isn’t the meaning of “a man”.

If anything, it translates the complete sentence

"There is a man"

Let’s try again, systematically…

**Why things get more complex**

- When doing predicate logic did you wonder why:
  - Kathy runs
    
    $\text{run}(\text{kathy})$
  - No kid runs
    
    $\neg (\exists x)(\text{kid}(x) \land \text{run}(x))$

- Somehow the NP’s meaning is wrapped around the predicate

- Or consider why this argument doesn’t hold:
  - Nothing is better than a life of peace and prosperity.
    
    $\neg (\exists x)(\text{life}(x, \text{peace and prosperity}))$
  - A cold egg salad sandwich is better than nothing.
    
    $\neg (\exists x)(\text{food}(x) \land \text{better}(x, \text{nothing}))$
  - The problem is that nothing is a quantifier

**Generalized Quantifiers**

- We have a reasonable semantics for red car in Palo Alto as a property from $\text{Ind} \rightarrow \text{Bool}$
- How do we represent noun phrases like the red car in Palo Alto or every red car in Palo Alto?
  - $\llbracket t \rrbracket(P) = a$ if $P(b) = 1$ iff $b = a$
  - undefined, otherwise
- The semantics for the following Bertrand Russell, for whom the $x$ meant the unique item satisfying a certain description

- red car in Palo Alto
  - select Cars.obj from Cars, Locations, Red where Cars.obj = Locations.obj AND Locations.place = 'paloalto' AND Cars.obj = Red.obj
  - (here we assume the unary relations have one field, obj).
Generalized Quantifiers

- the red car in Palo Alto
  \[ NP : \lambda x. \text{car}(x) \land \text{in}’(\text{paloalto})(x) \land \text{red}’(x) \]
  \[ \text{Det} : \lambda x. \text{car}(x) \land \text{in}’(\text{paloalto})(x) \land \text{red}’(x) \]
- the red car in Palo Alto
  \[ \text{select Cars.obj from Cars, Locations, Red where} \]
  \[ \text{Cars.obj = Locations.obj AND} \]
  \[ \text{Locations.place = ’paloalto’ AND Cars.obj = Red.obj having count(*) = 1} \]

Representing proper nouns with quantifiers

- The central insight of Montague’s PTQ was to treat individuals as of the same type as quantifiers (as type-raised individuals):  
  - Kathy : \( \lambda P. P(\text{kathy}) \)
  - Both good and bad
  - The main alternative (which we use) is flexible type shifting - you raise the type of something when necessary.
Noun phrase scope – following Hendriks (1993)

Value raising raises a function that produces an individual to one that produces a quantifier. If $\alpha : \sigma \rightarrow \text{Ind}$ then $\lambda x.\lambda P. P(\alpha(x)) : \sigma \rightarrow (\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{Bool}$

Argument raising replaces an argument of a boolean function with a variable and applies the quantifier semantically binding the replacing variable. If $\alpha : \sigma \rightarrow \text{Ind} \rightarrow \tau$ then $\lambda x_1.\lambda x_2.\lambda x_3. Q(\lambda x_1. \alpha(x_1)) (x_2) (x_3) : \sigma \rightarrow (\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \tau \rightarrow \text{Bool}$

Argument lowering replaces a quantifier in a boolean function with an individual argument, where the semantics is calculated by applying the original function to the type raised argument. If $\alpha : \sigma \rightarrow (\text{Ind} \rightarrow \text{Bool}) \rightarrow \text{bool} = \tau \rightarrow \text{Bool}$ then $\lambda x_1.\lambda x_2.\lambda x_3. \alpha(x_1) (x_2) (x_3) : \sigma \rightarrow \text{Ind} \rightarrow \tau \rightarrow \text{Bool}$

Some kid broke every toy

Questions with answers!

- A yes/no question (Is Kathy running?) will be something of type $\text{Bool}$, checked on database.
- A content question (Who likes Kathy?) will be an open proposition, that is something semantically of the type property ($\text{Ind} \rightarrow \text{Bool}$), and operationally we will consult the database to see what individuals will make the statement true.

We use a grammar with a simple form of gap-threading for question words

Syntax/semantics for questions

- $S' : \beta(\alpha) \rightarrow \text{NP[wh]} : \beta$ Aux $S : \alpha$
- $S' : \alpha \rightarrow \text{Aux} S : \alpha$
- $\text{NP/\text{NP}_2} : z \rightarrow e$
- $S : \lambda z. F(\ldots z \ldots) \rightarrow S/\text{NP}_2 : F(\ldots z \ldots)$
Syntax/semantics for questions

- **who**, NP[wh] : $MU.\lambda x.\text{U}(x) \land \text{human}(x)$
- **what**, NP[wh] : $\lambda U.\lambda x.\text{U}(x)$
- **which**, Det[wh] : $\lambda P.\lambda V.\lambda x.\text{P}(x) \land V(x)$
- **how many**, Det[wh] : $\lambda P.\lambda V.\lambda x.\text{P}(x) \land V(x)$

Where $\cdot|$ is the operation that returns the cardinality of a set (count).

Question examples

- $S' : \lambda z.\text{like}(z)(\text{kathy})$

  NP[wh] : $\lambda U.\lambda x.\text{U}(x)$
  Det : $\lambda P.\lambda V.\lambda x.\text{P}(x) \land V(x)$
  $\cdot| : \lambda P.\lambda V.\lambda x.\text{P}(x) \land V(x)$

  - select liked from Likes,Humans where Likes.liker='Kathy' AND Humans.obj = Likes.liked

  Question examples

- $S' : \lambda x.\text{car}(x) \land \text{like}(x)(\text{kathy})$

  NP[wh] : $\lambda U.\lambda x.\text{U}(x)$
  Det : $\lambda P.\lambda V.\lambda x.\text{P}(x) \land V(x)$
  $\cdot| : \lambda P.\lambda V.\lambda x.\text{P}(x) \land V(x)$

  - select from Cars,Likes,Locations,Reds where Cars.obj = Likes.liked AND Likes.liker = 'Kathy' AND Locations.place = 'Palo Alto' AND Locations.obj = Likes.liked
  - Did Kathy see the red car in Palo Alto?

  - select 'yes' where Seeings.seer = k AND Seeings.seen = (select Cars.obj from Cars, Locations, Red where Cars.obj = Locations.obj AND Locations.place = 'paloalto' AND Cars.obj = Red having count(*) = 1)
How could we learn such representations?

- After disengagement for many years, there has started to be very interesting work in this area:

How can we reason with such representations?

- Logical reasoning is practical for certain domains (business rules, legal code, etc.) and has been used (see Blackburn and Bos 2005 for background).
- But our knowledge of the world is in general incomplete and uncertain.
- There is various recent work on handling restricted fragments of first order logic in probabilistic models
- Undirected model:
- A recent attempt to apply this to natural language inference:
- Logical formulae are given weights which are grounded out in an undirected markov network