An Informal but Respectable Approach to Computational Semantics

CS224N/Ling 237 – Christopher Manning

May 23, 2000

1 Motivation

We have seen ways to extract and represent elements of meaning in statistical and other quantitative ways (word sense disambiguation, semantic similarity, information extraction templates). But there are many NLP domains, such as automated question answering, dialog systems, story understanding, and automated knowledge base construction, where we need to deal with the meanings of sentences in a more sophisticated yet robust way. Now what the *meaning* of a sentence is is a debatable issue, but the only thing implementation-ready as a system of representation seems to be the tradition of logical semantics, appropriately extended to natural language (Montague 1973). I believe that a promising approach for at least some applications is to use the benefits of statistical models for disambiguation at a lexical/syntactic level, and then to use logical semantics for detailed interpretations.

In computational semantics, we can isolate the following questions:

1. How can we automate the process of associating semantic representations with expressions of natural language?

2. How can we use semantic representations of natural language expressions to automate the process of drawing inferences?

We will concentrate on the first goal. This is for two reasons: firstly, this issue is normally ignored in logic classes, whereas the second is covered in depth. Secondly, because for present purposes, it's the mapping from language to a meaning representation that is most interesting. One way to proceed with a meaning representation is logical inference, but we can also explore quite different methods such as via probabilistic (relational) models.

The presentation here attempts to be self-contained, but basically assumes that you've seen some logic before: it covers some ideas that you don't normally see in a quarter course on logic/semantics.

The sentence meaning we will derive from the semantics discussed here has several distinguishing properties: it is *truth-conditional*, *compositional*, and *model-theoretic*. By *truth-conditional*, we mean that a sentence's meaning is what the world would have to be like in order for it to be true. By *compositional*, we mean that the meanings of phrases (including sentences) are generally built up from the meanings of the parts they contain, in a way that reflects the way in which those parts combine. And by *model-theoretic* (Tarski 1935), we mean that the logical language we use to build up meanings is distinguished from its *interpretation* in the domain of "real-world" objects and substantive relations.

2 Typed λ-calculus with a model-theoretic interpretation

We'll start with something like First-Order Predicate Calculus (FOPC), which contains constants like **j** and **k**, variables ranging over constants like x and y, and some n-ary functions which evaluate to truth values like **like**.

The secret weapon which we will use to automate the construction of meanings is the (typed) λ-calculus (Church 1940), which allows us to "hold out" positions within a FOPC expression and to "fill" them later, by applying the new expression to another expression. The fundamental rules to remember for λ-formulae are those that license *application* and *abstraction*, respectively:

$$(\lambda x.P(\cdots,x,\cdots))(Z) \quad\Rightarrow\quad P(\cdots,Z,\cdots) \qquad \beta \text{ reduction [application]}$$
$$\lambda x.(P(x)) \quad\Rightarrow\quad P \qquad \eta \text{ reduction [abstractions can be contracted]}$$

What's shown here is actually a somewhat complex process that includes conversion rules making sure variable names don't accidentally overlap, but we can ignore that for the moment.

Part of our semantics will include a mapping from syntactic units such as lexemes and categories to semantic expressions. In general, selectors such as verbs will be applied to their arguments, so we'll expect lexemes to look something like this:

John means **j** – a constant in the model.
Kathy means **k** – another constant.
likes means $\lambda y.\lambda x.\mathbf{like}(x,y)$

and syntactic rules to look something like this:

VP : $\alpha(\beta)$ → V : α NP : β

to mean that a VP expands as a V and an NP as before, but that the semantic translation is built up from the semantic translations of the daughters by function application in a rule-to-rule manner. Similarly, S : $\alpha(\beta)$ → NP : β VP : α. *Kathy* and *John* are of syntactic type NP, and *likes* is of syntactic type V. We have:

[$_{\text{VP}}$ likes Kathy] : $[\lambda y.\lambda x.\mathbf{like}(x,y)]k = \lambda x.\mathbf{like}(x,k)$ [β reduction]

[$_{\text{S}}$ John likes Kathy] : $[\lambda x.\mathbf{like}(x,k)]j = \mathbf{like}(j,k) = \mathbf{yes/no}$ [depending on model]

We will use a typed λ calculus, so we have λ terms of various types, such as **Bool** = truth values and **Ind** = individuals (more types are introduced later).

Church-Rosser theorem corollary: order of reductions of typed λ-calculus terms doesn't matter – and so there are usable proof-theoretic techniques (two terms are logically equivalent iff they are reducible to the same form). There are normal forms (the normal form uses η-reductions [removing lambdas] and α-reduction [renaming of variables]):

β,η normal form: **run**, **every**(**kid**, **run**), **yesterday**(**run**)
β,η long form: $\lambda x.\mathbf{run}(x)$, $\mathbf{every}(\lambda x.\mathbf{kid}(x))$, $(\lambda x.\mathbf{run}(x))$, $\lambda y.\mathbf{yesterday}(\lambda x.\mathbf{run}(x))(y)$

3 A simple example

Our goal is a system to build up a logical proposition out of the components of the sentence. We'll make use of the fact that syntax and compositional semantics, as traditionally studied, are actually pretty parallel:

(1) a. A bookshop is across the street.

b. A strange man is across the street.

The syntactic observation here is that [A bookshop] and [A strange man] are interchangeable without affecting the grammaticality of the sentence. Due in part to this observation, syntacticians posit that the two sequences of words are *constituents* of a common type–in this case, NP. The semantic observation is that the shared parts of these two sentences seem to have a common meaning, and that the contribution made by [A bookshop] and [A strange man] to sentence meaning is parallel.

What we'll do to capture this is to define a semantics in parallel to a syntax. In every syntactic rule, each category will be annotated with a semantic value, (typically) determining how the semantics of the mother is built out of the semantics of the daughters. Here's an example:

Kathy, NP : **kathy**	S : $\beta(\alpha)$ → NP : α VP : β
Fong, NP : **fong**	VP : $\beta(\alpha)$ →V : β NP : α
respects, V : **respect**	VP : β →V : β
runs, V : **run**	

kathy and **fong** are **Ind**
run is **Ind** → **Bool**
respect is **Ind** → **Ind** → **Bool**

With this syntax & semantics, we invariably get a logical proposition associated with the root node S of the sentence. Here's one of the six sentences in this language:

S : **respect**(**fong**)(**kathy**)
NP : **kathy** VP : **respect**(**fong**)
Kathy V : **respect** NP : **fong**
respects *Fong*

4 Logical Inference and Meaning Postulates

Once we have these predicate logic forms, we can do reasoning on them directly. Reasoning works pretty straightforwardly on the familiar propositional logic operators $\vee, \wedge, \Rightarrow, \neg$. For a system of meaning representation we might want to code in *meaning postulates* relating predicates to each other. For example, we might argue that

admire(x)(y) ⇒ **respect(x)(y)**

or, perhaps less controversially, if **give(x)(y)(z)** means "Z gives X Y", and **give(y)(z)** means "Z gives X", then

$$\mathbf{give(x)(y)(z)} \Rightarrow \mathbf{give(y)(z)}$$

5 Interpretation

So far we've got a *compositional* system of semantics, but you may be wondering, where is the truth-conditionality, or the model interpretation? We'll touch on that now.

We build a *model* $\mathcal{M} = \langle \mathbf{Dom}, [\![\cdot]\!] \rangle$ as a pair of a frame that gives a domain for basic types (with functional types defined in the obvious manner), and an interpretation function $[\![\cdot]\!] : \mathbf{Con} \to \mathbf{Dom}$, which tells us the denotation of semantic expressions. The denotation of a constant is taken to be an *individual*, whereas the denotation of a function is taken to be a *set*, and the meaning of a sentence is a truth value (Boolean). An assignment function similarly tells us the meaning of variables.

For our example in the previous section, we might use the following model:

$$[\![\mathbf{kathy}]\!] = k$$
$$[\![\mathbf{fong}]\!] = f$$
$$[\![\mathbf{run}]\!] = \{f\}$$
$$[\![\mathbf{respect}]\!] = \{(f,k),(k,k),(k,f)\}$$

The truth value of an n-ary function applied to an ordered n-tuple of constants is 1 if the n-tuple is in the function's meaning; it is 0 otherwise. The sentence we built in our previous section, then, has a truth value of 1, since (f,k) is in $[\![\mathbf{respect}]\!]$.

It is important to realize that this mapping is *not* trivial; it need not be 1-1. For example, it would be perfectly possible that $[\![\mathbf{kathy}]\!] = [\![\mathbf{fong}]\!] = k$, in the case that *Kathy and Fong were two different names for the same individual.* You can think of the values **kathy** and **fong** as being the more linguistic entities "that person who is referred to as Kathy (resp. Fong)." Much philosophical mileage may be gotten out of this distinction, and perhaps some computational mileage as well. The idea here would be to represent a database of knowledge in model-theoretic terms, and then evaluate the truth value of statements/queries (see a later section) directly against this database. However, in computational work there is often more focus on manipulation of the semantic values such as in the previous section, ignoring the model-theoretic interpretation.

6 Quantifiers and Higher order logic

"Seems pretty easy," you might say, "is that all there is to it?" Not quite. As soon as you get away from the domain of proper names and into noun phrases containing determiners or quantifiers, things start to change. When doing a predicate logic course, have you ever wondered how come *Kathy runs* is $\mathbf{run}(k)$, but *no kid runs* is $\neg(\exists x)(\mathbf{kid}(x) \wedge \mathbf{run}(x))$? Or you might have noticed that the following argument doesn't hold:

(2) Nothing is better than a life of peace and prosperity.
A cold egg salad sandwich is better than nothing.
____
A cold egg salad sandwich is better than a life of peace and prosperity.

although it would, given the semantics in the previous section, and assuming that the semantic rules for VP and S presented in the previous section applied straightforwardly to the semantic values for *nothing*, *a life of peace and prosperity*, and *a cold egg salad sandwich* (why?).

There are several ways to deal with quantifiers; the one we'll look at next time involves the use of a logic with a richer system of higher order *types*, among which *type-shifting* can occur. We will now allow variables and quantification over relations (over relations). But to keep track of things (and to avoid certain paradoxes), we'll keep track of the *type* of semantic values. We will use an ω-order logic (which allows any finite order quantification over types). Then we have λ terms of various types **Bool**, truth values; **Ind**, individuals; and higher-order functional types like **Ind** → **Bool** (which we call *properties*) and **Ind** → **Ind** → **Bool** – where we convert a several argument function into embedded unary functions by *currying* them. Our various functions have a definite type (e.g., **run** is a property of type **Ind** → **Bool**).

Assuming three individuals f, b, and r in our model, functions over individuals can be specified:

$$[\![\lambda x.\lambda y.\mathbf{respect}(y)(x)]\!] = \begin{bmatrix} f \mapsto \begin{bmatrix} f \mapsto \mathbf{no} \\ b \mapsto \mathbf{no} \\ r \mapsto \mathbf{yes} \end{bmatrix} \\ b \mapsto \begin{bmatrix} f \mapsto \mathbf{yes} \\ b \mapsto \mathbf{no} \\ r \mapsto \mathbf{yes} \end{bmatrix} \\ r \mapsto \begin{bmatrix} f \mapsto \mathbf{yes} \\ b \mapsto \mathbf{yes} \\ r \mapsto \mathbf{no} \end{bmatrix} \end{bmatrix}$$

$$[\![\lambda x.\lambda y.\mathbf{respect}(y)(x)]\!](b)(f) = \mathbf{yes}$$

Adjectives modify properties, and intensifiers modify adjectives, etc. E.g, *a not very happy camper*. We can question and quantify over many of these types.

Exercise: Assuming f and b are kids, what would **kid** be? How can logical constants like **and** and *quantifiers* like $\mathbf{every_{Ind}}$ be represented in this way? (Hint $\mathbf{every_{Ind}}$ is of type (**Ind** → **Bool**) → **Bool**.)

Crucially, now, we can perform λ-abstraction over arbitrary types, not just constants as we could before. This is necessary to capture the fundamental insight that the semantics of a quantifier or determiner contains the semantics of the verb, but the semantics of the verb contains the semantics of the noun next to the determiner. In order to get this result compositionally, a λ-expression must be applied to a non-constant expression. We'll see this next time.

But for now, we'll take a look at the nice fact that this higher-order type system allows us to do modification of VPs and NPs very easily. We expand somewhat the grammar from the previous section and take a look at the results.

Our major syntactic categories will have corresponding types: nouns and verb phrases will be properties (**Ind** →**Bool**), noun phrases are **Ind** – though, we will see later that they are commonly type-raised to (**Ind** → **Bool**) → **Bool**, adjectives are (**Ind** → **Bool**) → (**Ind** → **Bool**), etc.

The meaning of categories will be a lambda term. This can be thought of as just a symbolic stand-in for the domain element it denotes in some model, but for use of proof-theoretic techniques in computational applications, we need to be able to manipulate these terms.

In general, head-nonhead syntactic composition corresponds to functional application. However, in certain cases, semantic heads deviate from syntactic heads. This occurs principally with determiners, and modifiers (adjectives and adverbs).

Kathy, NP : **kathy**	$S : \beta(\alpha) \to NP : \alpha \quad VP : \beta$
Palo Alto, Det : **paloalto**	$NP : \beta(\alpha) \to Det : \beta \quad N' : \alpha$
car, N : **car**	$N' : \beta(\alpha) \to Adj : \beta \quad N : \alpha$
overpriced, Adj : **overpriced**	$N' : \beta(\alpha) \to N : \alpha \quad PP : \beta$
outside, PP : **outside**	$N' : \beta \to N : \beta$
red, Adj : $\lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$	$VP : \beta(\alpha) \to V : \beta \quad NP : \alpha$
in, P : $\lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$	$VP : \beta(\gamma)(\alpha) \to V : \beta \quad NP : \alpha \quad NP : \gamma$
the, Det : ι	$VP : \beta(\alpha) \to VP : \alpha \quad PP : \beta$
runs, V : **run**	$VP : \beta \to V : \beta$
respects, V : **respect**	$PP : \beta(\alpha) \to P : \beta \quad NP : \alpha$

in′ is **Ind → Ind → Bool** **run** is **Ind → Bool**

- S : $\mathbf{run}(\iota(\mathbf{overpriced}(\mathbf{car})))$
  - NP : $\iota(\mathbf{overpriced}(\mathbf{car}))$
    - Det : ι
      - *the*
    - N′ : $\mathbf{overpriced}(\mathbf{car})$
      - Adj : **overpriced**
        - *overpriced*
      - N : **car**
        - *car*
  - VP : **run**
    - V : **run**
      - *runs*

For intersective adjectives and prepositions we get spurious attachment ambiguities:

- N′ : $\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x))$
  - Adj : $\lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$
    - *red*
  - N′ : $\lambda x.(\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
    - N′ : **car**
      - N : **car**
        - *car*
    - PP : $\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
      - P : $\lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$
        - *in*
      - NP : **paloalto**
        - *Palo Alto*

- N′ : $\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x))$
  - N′ : $\lambda x.(\mathbf{car}(x) \wedge \mathbf{red'}(x))$
    - Adj : $\lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$
      - *red*
    - N′ : **car**
      - N : **car**
        - *car*
  - PP : $\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
    - P : $\lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$
      - *in*
    - NP : **paloalto**
      - *Palo Alto*

For non-intersective adjectives, we don't. We'll get:

overpriced(**in**(**paloalto**)(**house**))
in(**paloalto**)(**overpriced**(**house**))

Preposition modifying a VP:

- $\mathrm{S} : \mathbf{run}(\mathbf{kathy}) \wedge \mathbf{in'}(\mathbf{paloalto})(\mathbf{kathy}))$
  - $\mathrm{NP} : \mathbf{kathy}$
    - *Kathy*
  - $\mathrm{VP} : \lambda x.(\mathbf{run}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
    - $\mathrm{VP} : \mathbf{run}$
      - $\mathrm{V} : \mathbf{run}$
        - *runs*
    - $\mathrm{PP} : \lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x))$
      - $\mathrm{P} : \lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in'}(y)(x))$
        - *in*
      - $\mathrm{NP} : \mathbf{paloalto}$
        - *Palo Alto*

Note that because a VP and an N′ are both a property, a PP can modify either. On the other hand, note that the intersective semantics for *in* that I have used isn't necessarily correct here – it gives a "subject-centered" analysis rather than an "event-centered" analysis.

For such semantic forms, it is straightforward to evaluate them in a model or against a database or a knowledge base (which instantiates a model).