Computational Linguistics: Formal Semantics and Compositionality

Raffaella Bernardi

University of Trento

Contents

1. Seen to be done

We have seen:

- how lexical meaning is represented by sets.
- how to go from a set-theoretical representation to a functional one.
- the semantic types of the domain of interepretation.

Today we are going to introduce how to build the meaning representation of a sentence out of the meaning representation of words.

2. Reminder: Semantics is model-theoretic

The focus is on meaning as “extension”:

> “The extension of an expression is the set of things it extends to, or applies to” (Wikipedia)

Ingredients:

- A model of the world
- the model consists of sets
- words in a language refer or denote parts of the model
- a proposition is true iff it corresponds to state of affairs in the model.

3. Reminder: A Model of PL

A **model** consists of two pieces of information:

- which collection of atomic propositions we are talking about (**domain**, D),
- and for each formula which is the appropriate **semantic value**, this is done by means of a function called **interpretation function** ($\mathcal{I}$).

Thus a model $\mathcal{M}$ is a pair: $(D, \mathcal{I})$.

Correction of Exercises on set-theoretical vs functional meaning.

3.1. Frege: From words to sentences

Complete vs. Incomplete Expressions Frege made the following distinction:

- A sentence is a **complete** expression, it's reference is the truth value.
- A proper name stands for an object and is represented by a constant. It's a **complete** expression.
- A predicate is an **incomplete** expression, it needs an object to become complete. It is represented by a function. Eg. "left" needs to be completed by "Raj" to become the complete expression "Raj left".

Principle of Compositionality: The meaning of a sentence is given by the meaning of its parts and by the compositionality rules. This holds both at the reference and sense level.

4. Building Meaning Representations

To build a meaning representation we need to fulfill three tasks:

Task 1 Specify a reasonable **syntax** for the natural language fragment of interest.

Task 2 Specify semantic representations for the **lexical items**.

Task 3 Specify the **translation** of constituents **compositionally**. That is, we need to specify the translation of such expressions in terms of the translation of their parts, parts here referring to the substructure given to us by the syntax.

Moreover, when interested in Computational Semantics, all three tasks need to be carried out in a way that leads to computational implementation naturally.

We have looked at Task 1 in lecture 2 (formal grammars).

Today we will start looking at the other two tasks.

5. Lambda Calculus

FOL augmented with Lambda calculus can capture the "how" and accomplish tasks 2 and 3.

- It has a **variable binding operators** λ. Occurrences of variables bound by λ should be thought of as place-holders for missing information: they explicitly mark where we should substitute the various bits and pieces obtained in the course of semantic construction.
- An **operation** called β-conversion performs the required substitutions.

5.1. Lambda Calculus: Function and lambda terms

Function $f : X \to Y$. And $f(x) = y$ e.g. $SUM(x, 2)$ if $x = 5$, $SUM(5, 2) = 7$.

- $\lambda x.x$
- $\lambda x.(x + 2)$ [SUM(x,2)]
- $\underbrace{(\lambda x.(x + 2))}_{function} \underbrace{5}_{argument}$
- $\underbrace{(\lambda x.(x + 2))}_{function} \underbrace{5}_{argument} = 5 + 2$
- $((\lambda y.\lambda x.(x + y)) \underbrace{5}_{argument2}) \underbrace{2}_{argument1} = (\lambda x.(x + 5))\ 2 = 2 + 5$
- $\lambda y.\lambda x.(x + y) = \lambda(x, y).(x + y)$

5.2. Lambda-terms: Examples

Here is an example of lambda terms:

$$\lambda x.\texttt{left}(x)$$

The prefix $\lambda x.$ binds the occurrence of x in $\texttt{student}(x)$. We say it **abstracts** over the variable x. The purpose of abstracting over variables is to mark the slots where we want the substitutions to be made.

To glue `vincent` with “left” we need to apply the lambda-term representing “left” to the one representing “Vincent”:

$$\lambda x.\texttt{left}(x)(\texttt{vincent})$$

Such expressions are called **functional applications**, the left-hand expression is called the **functor** and the right-hand expression is called the **argument**. The functor is applied to the argument. Intuitively it says: fill all the placeholders in the functor by occurrences of the term `vincent`.

The substitution is performed by means of β-conversion, obtaining `left(vincent)`.

5.3. Functional Application

Summing up:

- FA has the form: Functor(Argument). E.g. $(\lambda x.love(x, mary))(john)$
- FA triggers a very simple operation: Replace the λ-bound variable by the argument. E.g. $(\lambda x.love(x, mary))(john) \Rightarrow love(john, mary)$

Exercise 3.

5.4. β-conversion

Summing up:

1. Strip off the λ-prefix,
2. Remove the argument,
3. Replace all occurences of the λ-bound variable by the argument.

For instance,

1. $(\lambda x.love(x, mary))(john)$
2. $love(x, mary)(john)$
3. $love(x, mary)$
4. $love(john, mary)$

5.5. Exercise

Give the lambda term representing a transitive verb.

(a) Build the meaning representation of “John saw Mary” starting from:

- John: `j`
- Mary: `m`
- saw: $\lambda x.\lambda y.\texttt{saw}(y, x)$

(b) Build the parse tree of the sentence by means the bottom-up method.

(c) Compare what you have done to assembly the meaning representation with the way you have built the tree.

5.6. α-conversion

Warning: Accidental bounds, e.g. $\lambda x.\lambda y.\texttt{Love}(y, x)(y)$ gives $\lambda y.\texttt{Love}(y, y)$. We need to rename variables before performing β-conversion.

α-conversion is the process used in the λ-calculus to rename bound variables. For instance, we obtain

$\lambda x.\lambda y.\texttt{Love}(y, x)$ from $\lambda z.\lambda y.\texttt{Love}(y, z)$.

When working with lambda calculus we always α-covert before carrying out β-conversion. In particular, we always rename all the bound variables in the functor so they are distinct from all the variables in the argument. This prevents accidental binding.

6. Lambda-Terms Interpretations

We've seen that a Model is a pair consisting of a domain ($\mathcal{D}$) and an interpretation function ($\mathcal{I}$).

- In the case of FOL we had only one domain, namely the one of the objects/entities we were reasoning about. Similarly, we only had one type of variables. Moreover, we were only able to speak of propositions/clauses.
- λ-terms speak of functions and we've used also **variables standing for functions**. Therefore, we need a more complex concept of interpretation, or better a more **complex concept of domain** to provide the fine-grained distinction among the objects we are interested in: truth values, entities and functions.
- For this reason, the λ-calculus is of Higher Order.

6.1. Models, Domains, Interpretation

In order to interpret meaning representations expressed in FOL augmented with λ, the following facts are essential:

- **Sentences**: Sentences can be thought of as referring to their truth value, hence they denote in the the domain $D_t = \{1, 0\}$.
- **Entities**: Entities can be represented as constants denoting in the domain D_e, e.g. $D_e = \{\mathtt{john}, \mathtt{vincent}, \mathtt{mary}\}$
- **Functions**: The other natural language expressions can be seen as incomplete sentences and can be interpreted as **boolean functions** (i.e. functions yielding a truth value). They denote on functional domains $D_b^{D_a}$ and are represented by functional terms of type $(a \to b)$.

  For instance "walks" misses the subject (of type e) to yield a sentence (t).

  - denotes in $D_t^{D_e}$
  - is of type $(e \to t)$,
  - is represented by the term $\lambda x_e(\mathtt{walk}(x))_t$

6.2. Lambda-calculus: some remarks

The pure lambda calculus is a theory of functions as rules invented around 1930 by Church. It has more recently been applied in Computer Science for instance in "Semantics of Programming Languages".

In Formal Linguistics we are mostly interested in lambda conversion and abstraction. Moreover, we work only with typed-lambda calculus and even more, only with a fragment of it.

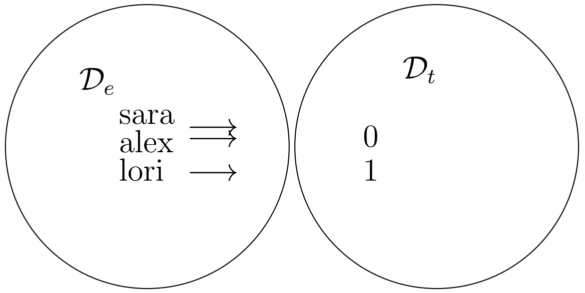
The types are the ones we have seen above labeling the domains, namely:

- e and t are types.
- If a and b are types, then $(a \to b)$ is a type.

7. Lambda Terms represent functions

For instance "walk" is a set of entities (those entities which walk), hence it's a function:

- denotes in $D_t^{D_e}$
- is of type $(e \to t)$,
- is represented by the term $\lambda x_e(\texttt{walk}(x))_t$



$[\![walk]\!] = \{lori\}$

7.1. Exercises: Lambda terms

1. Build a model for a situation of your choice. Specify the domains of interpretation, the set-theoretical representation of words, and their corresponding typed lambda terms.
2. If an intransitive verb (e.g. "walk") is represented by a unary-function, a transitive verb (e.g. "know") by a binary-function, what is the function representing a ditransitive verb (e.g. "gave")?
3. What could be the meaning representation of an adjective (e.g. "red")?