

Formal Semantics: A Brief Re-introduction

Curt Anderson
Heinrich-Heine-Universität Düsseldorf

October 14, 2019

1 Semantics and the model

In this class, we adopt a view that the meaning of an expression can be captured by considering its truth conditions, what it takes to make a particular sentence true or false.

Truth is relative to a model, a list of the things and how they relate to each other.

Two basic types in the model: the type of “entities” (e) and the type of truth-values (t). These are sets. D_e is the set of all entities, and D_t is the set of truth values (of which there are only two, 0 for ‘false’ and 1 for ‘true’).

- (1) a. D_e
- b. $D_t = \{0, 1\}$

2 Functions

Principle of compositionality: The meaning of a linguistic expression is determined by the combination of its immediate subparts.

In order to build up the meaning of a sentence compositionally, we represent the meaning of some linguistic expressions as functions. A function simply takes an input and gives back some output.

A classic way of thinking about functions is that they are expressions with a hole in them. When this hole is unfilled, functions are said to be unsaturated, and when it is filled, they are said to be saturated. The way we mark that some logical expression has a hole in it is with a λ . The λ binds a variable in its scope, showing that that variable in its scope is unvalued and needs to be valued. Variables with λ s binding them are called arguments.

- (2) $\lambda x[\dots \text{scope} \dots]$

Here is an example of a function that takes a number as an input, and returns a truth value; this function returns 1 just in case the input x added to 2 is equal to 4. (So, this function is

true if the input is 2.)

- (3) $\lambda x[1 \text{ iff } x + 2 = 4]$

If it helps, you can break it up into an input and an output.

- (4) a. Input: a number x
- b. Output: The function is true iff $x + 2 = 4$

Or, a more linguistic example. The denotation brackets ($\llbracket \cdot \rrbracket$) We can simplify the bit in the square brackets to use predicate logic. The 1 iff part can also be dropped, since it is redundant. (We’ll treat **dog** as if it is a black box; individuals go in, truth comes out, but we don’t quite know how.)

- (5) a. $\llbracket \text{dog} \rrbracket = \lambda x_e[1 \text{ iff } x \text{ is a dog}]$
- b. $\llbracket \text{dog} \rrbracket = \lambda x_e[\text{dog}(x)]$

To show that some value is the input to a function, the value is placed in parentheses after the function:

- (6) $\lambda x[1 \text{ iff } x + 2 = 4](2)$

More complex functions are built up by having functions that give you back another function.¹ Here is a mathematical function that takes two arguments and adds them together, returning the result. Note that this doesn’t have a 1 iff in it; this is because we are not outputting a truth value. ($x + y$ is a number, not a truth value.)

- (7) $\lambda x[\lambda y[x + y]]$

For instance, we can take the verb *kick* as a function that takes an individual (the direct object argument) and then returns a function that needs an argument (the subject argument).

- (8) $\llbracket \text{kick} \rrbracket = \lambda x[\lambda y[y \text{ kicks } x]]$

When saturating the arguments, we thus start with the “outermost” argument and work “inward”. Remember to drop off the λ and the variable once you have saturated the argument.

- (9) a. $\llbracket \text{kick} \rrbracket(\text{John})(\text{Mary})$

¹This way of building up multi-argument expressions by returning functions is called currying or Schönfinkelization, depending on who you ask.

- b. $\lambda x[\lambda y[y \text{ kicks } x]](\text{John})(\text{Mary})$
- c. $\lambda y[y \text{ kicks John}](\text{Mary})$
- d. **Mary** kicks **John**

Oftentimes, we leave off extra brackets when using multiple lambdas, or abbreviate the entire scope marked with the brackets with a single dot. The following are all equivalent.

- (10)
- a. $\lambda x[\lambda y[\lambda z[z \text{ gave } z \text{ to } x]]]$
 - b. $\lambda x\lambda y\lambda z[z \text{ gave } z \text{ to } x]$
 - c. $\lambda x\lambda y\lambda z.z \text{ gave } z \text{ to } x$

Finally, sometimes we may annotate the type of the variable after the λ for more clarity.

- (11)
- a. $\lambda f_{\langle e, t \rangle} \lambda x_e . f(x)$
 - b. $\lambda g_{\langle e, et \rangle} [\lambda x_e [\lambda y_e [g(y)(x)]]]$
 - c. $\lambda x_e \lambda y_e . \text{see}(y)(x)$

3 Types

We already discussed the basic types, type e and type t . Functions allowed us to chain together mappings between types, in order to create more complex expressions with multiple arguments.

To show that we have a mapping from one type to another, we use angle brackets ($\langle \cdot \rangle$). The type before the comma is the type of the input to the function, and the type after the comma is the type of thing the function outputs. Remember that since multi-argument functions are built up from simpler one-argument functions that spit out another function, we'll have types that are functions that return functions.

- (12)
- a. $\llbracket \text{dog} \rrbracket = \lambda x . \text{dog}(x)$
 - b. $\langle e, t \rangle$

- (13)
- a. $\llbracket \text{give} \rrbracket = \lambda x [\lambda y [\lambda z [z \text{ gave } z \text{ to } x]]]$
 - b. $\langle e, \langle e, \langle e, t \rangle \rangle \rangle$

Types can be arbitrarily fancy.

- (14)
- a. $\langle \langle e, t \rangle, \langle e, t \rangle \rangle$
 - b. $\langle e, e \rangle$
 - c. $\langle \langle e, t \rangle, t \rangle$

Note on conventions: oftentimes, if the type is complex, we might abbreviate it slightly by removing brackets and commas. Examples:

- (15)
- a. $\langle e, \langle e, t \rangle \rangle \rightarrow \langle e, et \rangle$
 - b. $\langle \langle e, t \rangle, \langle e, t \rangle \rangle \rightarrow \langle et, et \rangle$
 - c. $\langle e, \langle e, \langle e, t \rangle \rangle \rangle \rightarrow \langle e, eet \rangle$
 - d. $\langle \langle e, \langle e, t \rangle \rangle, t \rangle \rightarrow \langle et, t \rangle$

Naturally, functions are part of the model. Our model will have domains for the different functions. Examples:

- (16)
- a. $D_{\langle e, t \rangle} = \{\text{dog, run, ...}\}$
 - b. $D_{\langle e, et \rangle} = \{\text{kick, eat, ...}\}$
 - c. ...

How to determine type:

1. If helpful for you, first add brackets to show the scope of each lambda.

- (17)
- a. $\lambda x \lambda y \lambda z . \text{give}(x, y, z)$
 - b. $\lambda x [\lambda y [\lambda z [\text{give}(x, y, z)]]]$

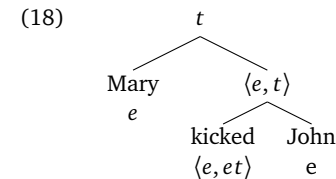
2. Working from the outermost scope towards the innermost scope, first determine the type of the argument.
3. Then, determine the type of the scope. If the scope is itself a function, repeat the previous step.

Some names for particular types:

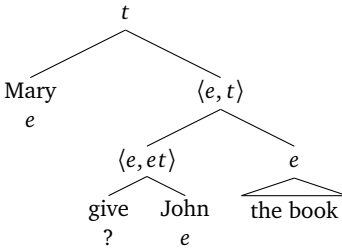
- Type t often called a truth value.
- Type e often called an individual.
- Type $\langle e, t \rangle$ is often called a property or a predicate. (Property of individuals.)
- Type $\langle e, et \rangle$ is often called a relation. (Relation between individuals.)

4 Type-annotated trees

The order of composition of arguments can be visualized using a tree structure. For each pair of branches in the tree, one branch must be an argument of the other branch.



Can infer types based on the types of things you already know and comparing the function/argument relations in the tree.

- (19) a. 
- b. $\llbracket \text{give} \rrbracket$ is type $\langle e, \langle e, et \rangle \rangle$

5 Other logical symbols

Besides λ , there are other logical symbols we will regularly encounter in class. First, there are \wedge and \vee , the logical connectives. These have roughly the meaning of *and* and *or*. They are used to conjoin two expressions of type t .²

There is also negation, \neg . This reverses a truth value.

- (20) a. $\neg \text{kissed}(\text{John})(\text{Bessie})$
 b. $\neg p$, where p is a proposition.

There are also the logical quantifiers, \forall and \exists . These quantifiers bind a variable, and the scope is used to set a condition on that variable.

- \forall says that all valuations of the bound variable must adhere to the condition.
- \exists says that at least one valuation of the bound variable must adhere to the condition.

Example:³

- (21) a. $\llbracket \text{Every dog bit Curt} \rrbracket = \forall x[\text{dog}(x) \rightarrow \text{bite}(x)(\text{Curt})]$
 b. $\llbracket \text{A dog bit Curt} \rrbracket = \exists x[\text{dog}(x) \wedge \text{bite}(x)(\text{Curt})]$
 c. $\llbracket \text{No dogs bit Curt} \rrbracket = \neg \exists x[\text{dog}(x) \wedge \text{bite}(x)(\text{Curt})]$

Important! $\exists x[\dots x \dots]$ isn't an individual itself! This is a truth value, type t ! The following is not well-formed!

- (22) $\text{kick}(\text{John})(\exists x[\text{ball}(x)])$ (NO!)
 intended: John kicked a rock.

Need to have the existential quantifier take scope over the entire expression.

- (23) $\exists x[\text{kick}(\text{John})(x) \wedge \text{rock}(x)]$
 There is an x such that John kicked x , and x is a rock.

There is one way of introducing an individual: using the ι operator (essentially “the”). This introduces an expression of type e . ι presupposes that there is a unique individual in the context that satisfies the condition in its scope for the bound variable.

- (24) $\iota x.\text{dog}(x)$

Naturally, since this is type e , it can be used anywhere as a regular type e expression.

- (25) $\llbracket \text{The dog bit Curt} \rrbracket = \text{bite}(\iota x.\text{dog}(x))(\text{Curt})$

Last note: when trying to translate into the logic, it often helps to say what you intend to mean in prose, and then try to rephrase it using the logic. It's often much easier to say what you mean than to state it using the formalization. For instance, this is ok to say as a first step.

- (26) a. $\llbracket \text{A dog bit Curt} \rrbracket = \text{There's a dog } x \text{ such that } x \text{ bit Curt.}$
 b. $\llbracket \text{dog} \rrbracket = \text{give me an individual } x \text{ and I'll tell you whether } x \text{ is a dog (or not)}$
 c. $\llbracket \text{mother} \rrbracket = \lambda x \lambda y. y \text{ is the mother of } x$

²This is important to note, since natural language *and* is much freer. It conjoins things of any type.

³You might be wondering why the logical connection changes from implication (\rightarrow) to conjunction (\wedge). The implication says that, if the antecedent (the bit before the connective) is true, then the consequent must be true. The conjunction says that both conjuncts are true. If (21-a) used a conjunction, then it would mean that everything is a dog and it bit Curt, which is too strong.

6 Exercises

6.1 From descriptions to lambda expressions

Express each of these functions, described here in words (in several different ways), using the λ notation.⁴

- (27)
- the function from individuals to truth values that applies to an individual and yields 1 iff the individual is a chimpanzee.
 - the function from individuals to truth values that applies to an individual and yields 1 iff the individual likes chimpanzees
 - the function from individuals to truth values that applies to an individual and yields 1 iff the individual is Portuguese
 - the function that maps an individual to 1 iff that individual is Herman
 - the function that maps an individual x to 1 iff x is afraid of chimpanzees
 - the function that maps an individual x to 0 iff x is not afraid of chimpanzees
 - the function that maps an individual x to 0 iff x is a cow
 - the function that is satisfied by individuals that have trouble sleeping
 - the function that is satisfied by anyone Portuguese
 - the function that holds of Herman and his monkey and of no other individual

6.2 Lambda conversion

Convert the following lambda expression to simplify them as much as possible. Remember to keep track of the brackets.

- (28)
- $[\lambda x[\lambda y[y \text{ kissed } x]]](\text{Bessie})$
 - $[\lambda x\lambda y[\lambda z[z \text{ kissed } x \text{ and } z \text{ hates } y]]](\text{Bessie})(\text{Clyde})$
 - $\lambda y[y \text{ eats grass and } [\lambda x[x \text{ is a cow}]](y) = 1]$
 - $[\lambda x[\lambda y[\lambda z[x = y \text{ and } z = x]]]](3)$

6.3 Fancier types

Identity the type of each of these functions. In some cases it will be ludicrously high, that is, complicated. You may have to do some lambda conversion first.

- (29)
- $\lambda f_{(e,t)}[f(\text{Herman}) = 1]$
 - $\lambda f_{(e,t)}[\lambda x[f(x) = 1]]$
 - $\lambda x[\lambda f_{(e,\langle e,t \rangle)}[f(\text{Clyde})(x) = 1]]$
 - $[\lambda x[\lambda f_{(e,\langle e,t \rangle)}[f(\text{Clyde})(x) = 1]]](\text{Herman})$
 - $\lambda f_{(e,t)} \left[\lambda x \left[\lambda y \left[\lambda z \left[\begin{array}{l} f(x) = 1 \wedge \\ f(y) = 1 \wedge \\ f(z) = 1 \end{array} \right] \right] \right] \right]$

$$\begin{aligned} \text{f. } & \left[\lambda f_{(e,t)} \left[\lambda x \left[\lambda y \left[\lambda z \left[\begin{array}{l} f(x) = 1 \wedge \\ f(y) = 1 \wedge \\ f(z) = 1 \end{array} \right] \right] \right] \right] \right] (\llbracket \text{whimper} \rrbracket)(\text{Greta}) \\ \text{g. } & \left[\lambda x \left[\lambda f_{(e,t)} \left[\lambda y \left[\begin{array}{l} f(x) = 1 \wedge \\ f(y) = 1 \end{array} \right] \right] \right] \right] (\text{Floyd})(\lambda z[z \text{ whimpered}]) \end{aligned}$$

⁴A function is *satisfied by* or *holds of* x if, when it applies to x , it yields 1.