LING4400: Lecture Notes 2
Simple Formal Logic

Last time we looked at inferences over sentence meanings. Now we'll see where they come from.

We'll build sentence meanings in a simple and expressive **higher-order logic** [Church, 1940].

Contents

2.1 Types of mental objects [Church, 1940]

We'll formalize sentence meanings using entities, truth values and functions of different **types**

(these are **mental objects** we can define associations over, and we'll use **bold font** for them here):

1. **entities** (type 'e'): what associations can be *about* (countries, people, units of volume, ...).

   For example, a small geographical domain may have two entities: **Laos** and **Togo**.

2. **truth values** (type 't'): values indicating belief or disbelief in certain kinds of associations.

   For example, here are the standard truth values we use: **False** and **True**.

3. **functions** (type '$\langle\alpha,\beta\rangle$'): map instances of **input** type α to instances of **output** type β.

   For example, an '*Is it coastal?*' function from e to t (as spreadsheet table): .

   input output
   Laos : **False**
   Togo : **True**

   These functions are associations listeners make in their minds, e.g.: **Togo** evokes **True**.

Functions may have *other functions* as inputs or outputs, yielding an unlimited number of types:

$\langle e, t\rangle$	maps entities to truth values
$\langle e, \langle e, t\rangle\rangle$	maps entities to other functions which then map entities to truth values
$\langle\langle e, t\rangle, t\rangle$	maps functions that map from entities to truth values to other truth values
$\langle e, \langle\langle e, t\rangle, t\rangle\rangle$	...

Functions with other functions as input or output will have nested associations, like Russian dolls.

For example, this function of type $\langle \mathrm{e}, \langle \mathrm{e}, \mathrm{t}\rangle\rangle$ maps entities to functions from entities to truth values:

<table>
<tr><th>input</th><th>output</th></tr>
<tr><td>Laos :</td><td><table><tr><th>input</th><th>output</th></tr><tr><td>Laos :</td><td>True</td></tr><tr><td>Togo :</td><td>False</td></tr></table></td></tr>
<tr><td>Togo :</td><td><table><tr><th>input</th><th>output</th></tr><tr><td>Laos :</td><td>False</td></tr><tr><td>Togo :</td><td>True</td></tr></table></td></tr>
</table>

Functions must be defined for *all* instances of the *input* type, so tables for e.g. $\langle\langle \mathrm{e}, \mathrm{t}\rangle, \mathrm{t}\rangle$ can get *big*!

That's because there are o^i instances of functions from i inputs to o outputs. (8 $\langle \mathrm{e}, \mathrm{t}\rangle$'s if 3 e's, 2 t's.)

(Functions with functions as input are complicated, so we'll look at logics with limited inputs first.)

Practice 2.1:

How many functions of type $\langle \mathrm{e}, \mathrm{t}\rangle$ are there in a world with two e's: (**A**, **B**), and two t's?

Practice 2.2:

List all the possible functions of type $\langle \mathrm{e}, \mathrm{t}\rangle$ in a world with two e's: (**A**, **B**), and two t's.

2.2 World models: collections of listeners' associations

Formally, we model language as transmitting associations from speakers' to listeners' minds.

We model listeners'/speakers' minds as **world models** – collections of associations about the world.

A world model M defines:

1. a **domain** D_α^M for each type α – a (possibly infinite) set of instances of that type in M;

   for example, in our geographical model, the domain of entities D_e^M would be **Laos** and **Togo**

   (domains for functions are all possible mappings between domains of input and output types);

2. an **interpretation function** $[\![\varphi]\!]^M$ – associating logical expressions φ into these domains;

   for example, the interpretation $[\![\mathsf{Coastal}]\!]^M$ would be the association:

input	output
Laos :	**False**
Togo :	**True**

   .

   We mostly define functions in world models and get sentence meanings via composition rules.

An interpretation function is itself an association from logical expressions to mental objects. World models are **complete**. Listeners with incomplete knowledge consider *multiple* world models.

A point about defining formal languages:

- stuff *inside* '⟦' and '⟧' brackets is the **formal language** we're defining;
- stuff *outside* brackets is a **metalanguage** we're using to define it: tables, pictures, words, etc. *except* sometimes we use **metavariables** (φ,χ,ψ,ω) *inside* brackets to generalize our claims.

2.3 Logical expressions: representations of ideas (and how to build them)

Logical expressions (what φ ranges over, above) are what interpretation functions interpret. The output of an interpretation function is called a **denotation** or **extension**.

We'll start with some **basic** expressions, which consist of just a single term:

1. If φ is a **constant**, the interpretation is defined by the world model.

   For example, here are some constant expressions of type e (we'll use this font for constants):

   $$[\![\mathsf{Laos}]\!]^M = \textbf{Laos}\,,$$
   $$\underbrace{[\![\mathsf{Togo}]\!]}_{\text{expression}}{}^M = \underbrace{\textbf{Togo}}_{\text{mental object}},$$

   (we'll let the name of every entity in a world model be a constant denoting that entity);

   and here's a constant function expression of type $\langle \mathrm{e}, \mathrm{t} \rangle$:

   $$\underbrace{[\![\mathsf{Coastal}]\!]}_{\text{expression}}{}^M = \underbrace{\boxed{\begin{array}{l} \text{input} \quad \text{output} \\ \hline \textbf{Laos} : \textbf{False} \\ \textbf{Togo} : \textbf{True} \end{array}}}_{\text{mental object}}\,.$$

   (We'll try to keep constants close to words, but understand they *only mean one thing*.)

2. Expressions can also be **variables** if bound by a lambda abstraction, discussed below.

If φ is not a constant or variable, the interpretation is **compositional** (depends entirely on its parts). We'll need only two kinds of compositional rules in our logical expressions. These will:

1. **apply** functions ψ of type $\langle\alpha,\beta\rangle$ to **arguments** χ of type α to get output $\omega = \psi\;\chi$ of type β;

this just looks up the output ω corresponding to the input $[\![\chi]\!]^M$ in the function (table) $[\![\psi]\!]^M$:

$$[\![\underbrace{\psi\ \chi}_{\text{expression}}]\!]^M = \underbrace{\omega \text{ such that } [\![\psi]\!]^M = \boxed{\begin{array}{cc} \text{input} & \text{output} \\ \vdots & \vdots \\ [\![\chi]\!]^M \ : & \omega \\ \vdots & \vdots \end{array}}}_{\text{mental object}},$$

for example, applying $\mathsf{Coastal}$ (type $\langle \mathrm{e},\mathrm{t}\rangle$) to Laos (type e) looks up **Laos** in $[\![\mathsf{Coastal}]\!]^M$:

$$[\![\mathsf{Coastal\ Laos}]\!]^M = \textbf{False}$$

(here '$\mathsf{Coastal}$' is the ψ, 'Laos' is the χ, '**Laos**' is the $[\![\chi]\!]^M$, and '**False**' is the ω);

2. **abstract** expressions ψ of type β over **variables** χ of type α to get functions $\lambda_{\chi:\alpha}\ \psi$ of type $\langle\alpha,\beta\rangle$; this creates a new function (table) with input drawn from the domain D^M_α of type α:

$$[\![\underbrace{\underbrace{\lambda_{\chi:\alpha}}_{\text{input of type }\alpha} \quad \underbrace{\psi}_{\text{output of type }\beta}}_{\text{expression of type }\langle\alpha,\beta\rangle}]\!]^M = \underbrace{\boxed{\begin{array}{ll} \text{input} & \text{output} \\ \iota_1 & : [\![\psi \text{ but replacing } \chi \text{ with } \varphi_1 \text{ where } [\![\varphi_1]\!]^M = \iota_1]\!]^M \\ \iota_2 & : [\![\psi \text{ but replacing } \chi \text{ with } \varphi_2 \text{ where } [\![\varphi_2]\!]^M = \iota_2]\!]^M \\ \vdots & \qquad\qquad\vdots \end{array}} \text{ for } \iota_1, \iota_2, \ldots \text{ in } D^M_\alpha}_{\text{mental object}},$$

for example, abstracting $\mathsf{Coastal}\ x$ (type t) over variable x (type e) gives an $\langle \mathrm{e},\mathrm{t}\rangle$ function:

$$[\![\ \underbrace{\underbrace{\lambda_{x:\mathrm{e}}}_{\text{input of type e}} \ \underbrace{\mathsf{Coastal}\ x}_{\text{output of type t}}}_{\text{expression of type }\langle \mathrm{e},\mathrm{t}\rangle}\ ]\!]^M = \boxed{\begin{array}{ll} \text{input} & \text{output} \\ \textbf{Laos} : & \textbf{False} \\ \textbf{Togo} : & \textbf{True} \end{array}}$$

(you will also sometimes see $\lambda_{\chi:\alpha}\ \psi$ notated with a dot after the variable: $\lambda\chi{:}\alpha\ .\ \psi$).

Variables can only appear in the ψ of the lambda subscripted by that variable.

Another way to think of these abstractions is as **function definitions**, as used in algebra:

$$f(x) = \mathsf{Coastal}\ x \qquad \text{or} \qquad f\ x = \mathsf{Coastal}\ x$$

except they define only the function, so the argument is moved to the other side of the equals:

$$f = \lambda_{x:\mathrm{e}}\ \mathsf{Coastal}\ x$$

To be strict, we should also add a composition rule to:

3. **parenthesize** expressions ψ, in order to group their function applications:

$$[\![(\psi)]\!]^M = [\![\psi]\!]^M$$

   This doesn't do anything itself, but helps us tell which functions to apply in what order.

   For example, the two parenthesizations below mean different things:

$$[\![(\text{Not Erupt})\ \text{Twice}]\!]^M \neq [\![\text{Not}\ (\text{Erupt Twice})]\!]^M$$

   The first expression (*‘two lulls’*) applies Not to Erupt, then applies the result to Twice.

   The second expression (*‘<2 eruptions’*) applies Erupt to Twice, then applies Not to the result.

   With no parens, we assume the first grouping (so application is **left-to-right associative**):

$$[\![\text{Not Erupt Twice}]\!]^M = [\![(\text{Not Erupt})\ \text{Twice}]\!]^M$$

   But abstractions group the lambda operators last (so abstraction is **right-to-left associative**):

$$[\![\lambda_{x:\mathrm{e}}\ \lambda_{y:\mathrm{e}}\ \text{Contain}\ x\ y]\!]^M = [\![\lambda_{x:\mathrm{e}}\ (\lambda_{y:\mathrm{e}}\ (\text{Contain}\ x\ y))]\!]^M$$

These kinds of expressions are called **lambda calculus**, because of all the lambdas.

Practice 2.3:

Write a lambda calculus function that multiplies a number by two and then adds one. You can use the symbols ‘×’ and ‘+’ inside your function.

Practice 2.4:

Write a lambda calculus expression that applies your function above to the number 3. You don't have to show the result.

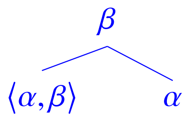
2.4 Drawing expressions as trees

We can draw lambda calculus expressions as trees to see how they are constructed.

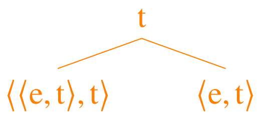
These will look like family trees with (always single) parents on top and children below.

Basic expressions like **constants** and **variables** will have no children. As for the others:

1. we draw **applications** as branches from a parent of type β to children $\langle\alpha,\beta\rangle$ and α:

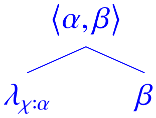


for example:

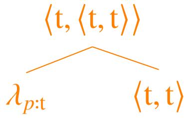


where $\alpha = \langle \mathrm{e},\mathrm{t}\rangle$ and $\beta = \mathrm{t}$.

2. we draw **abstractions** as branches from parent $\langle\alpha,\beta\rangle$ to children $\lambda_{x:\alpha}$ and β:



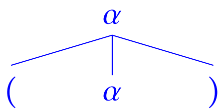
for example:



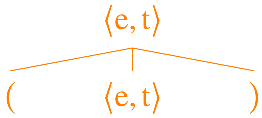
where $\alpha = \mathrm{t}$ and $\beta = \langle \mathrm{t}, \mathrm{t}\rangle$.

Remember: the $\lambda_{\chi:\alpha}$ is not a variable and has no type — it's a literal lambda in the expression.

3. we draw **parentheses** as branches from a parent of type α to children (, α and):

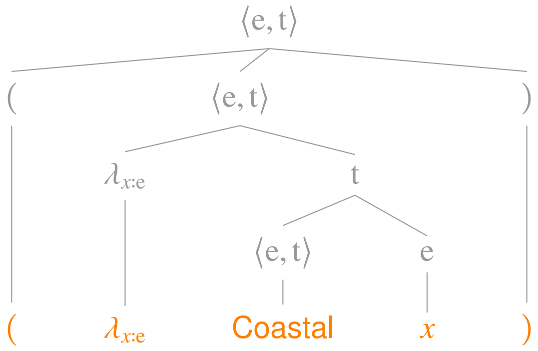


for example:



where $\alpha = \langle \mathrm{e}, \mathrm{t}\rangle$.

As a complete example, the derivation tree for $(\lambda_{x:\mathrm{e}}\ \mathsf{Coastal}\ x)$ is:



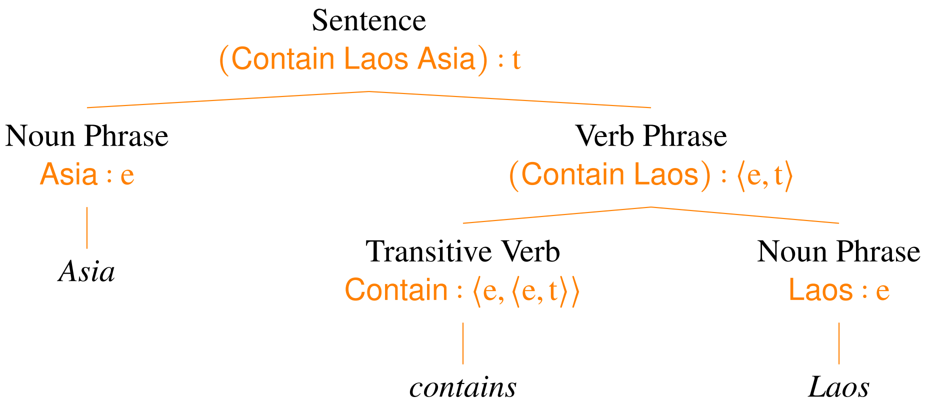
From top to bottom, we have a parentheses on top, then an abstraction, and an application below.

We will call these **derivation trees**, or 'gray trees'. They have basic expressions as leaves.

2.5 Using composition rules to obtain sentence meanings

We formally translate English sentences into logic expressions by associating words with functions.

We use variants of composition rules when we assemble word meanings into sentence meanings:



Here, words corresponding to basic expressions (constants) are composed into phrases and clauses.

Note the top branch applies a function on the *right* with an argument on the *left*. That's backward!

In building logic translations for sentences, this is called **Backward Function Application**:

$$g : \alpha \quad f : \langle \alpha,\beta \rangle \;\Rightarrow\; (f\ g) : \beta \qquad \text{(Backward Function Application)}$$

We use this variant together with normal **Forward Function Application** in translating to logic:

$$f : \langle \alpha,\beta \rangle \quad g : \alpha \;\Rightarrow\; (f\ g) : \beta \qquad \text{(Forward Function Application)}$$

These rules say if you combine the two child expressions on the left you get the parent on the right.

We will call these additional composision rules **translation rules**.

They define how to build trees like the one above from the bottom up.

We will call these resulting trees **translation trees** or 'orange trees'. They have words as leaves.

2.6 Rules for simplifying expressions

There are also **reduction rules** we use to simplify lambda calculus expressions. They ensure:

1. The denotation of a function f is the same as that of $\lambda_{x:\alpha}\ f\ x$ in all world models M:

   $$[\![f]\!]^M = [\![\lambda_{x:\alpha}\ f\ x]\!]^M .$$

   For example:

   $$[\![\text{Coastal}]\!]^M = [\![\lambda_{x:\text{e}}\ \text{Coastal}\ x]\!]^M = \begin{array}{|l|} \hline \text{input} \quad \text{output} \\ \hline \textbf{Laos} : \textbf{False} \\ \textbf{Togo} : \textbf{True} \\ \hline \end{array} .$$

   This is called **eta conversion** or **η-conversion**.

2. A function abstraction also denotes the same if all instances of the variable are renamed:

   $$[\![\lambda_{x:\alpha} \ldots x \ldots x \ldots]\!]^M = [\![\lambda_{y:\alpha} \ldots y \ldots y \ldots]\!]^M .$$

   This is called **alpha conversion** or **α-conversion**.

3. A function application is the same as the function with no lambda and argument substituted:

   $$[\![(\lambda_{x:\alpha} \ldots x \ldots x \ldots)\ a]\!]^M = [\![\ldots a \ldots a \ldots]\!]^M .$$

   (We remove the *first* lambda and replace *all* instances of that variable with the *first* argument.)

   For example:

   $$[\![(\lambda_{x:\text{e}}\ \text{Contains}\ x\ \text{Asia})\ \text{Laos}]\!]^M = [\![\text{Contains Laos Asia}]\!]^M = \textbf{True}.$$

   This is called **beta reduction** or **β-reduction**.

These rules let us simplify lambda calculus expressions without calculating denotations.

Practice 2.5:

Beta reduce the following expression:

$$(\lambda_{x:\mathrm{e}}\ (\mathsf{And}\ (\mathsf{Coastal}\ x)\ (\mathsf{Capital}\ x)))\ \mathsf{Laos}$$

Practice 2.6:

Beta reduce the following expression:

$$(\lambda_{y:\mathrm{e}}\ \lambda_{x:\mathrm{e}}\ \mathsf{Contain}\ y\ x)\ \mathsf{Laos}\ \mathsf{Asia}$$

References

[Church, 1940] Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2), 56–68.