CITS3005 Knowledge Representation

Lecture 9: Ontologies

Adapted from Hogan et al's Knowledge Graphs
https://krbook.org
and Jean Baptiste Lamy's Ontologies with Python
https://doi.org/10.1007/978-1-4842-6552-9

The University of Western Australia

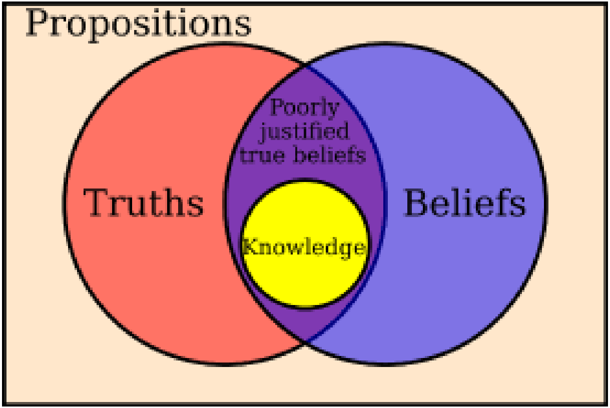
2023

Knowing what vs knowing how

So far, our study of knowledge graphs has focused on *knowing what*. That is, we assemble facts in some logical structure from which we ask queries.

However, if we would like to ask *how* or *why* things are true, we need to understand the causal and abstract relationships surrounding the data.

Ontologies are needed to represent these relationships.



Ontologies (informally)

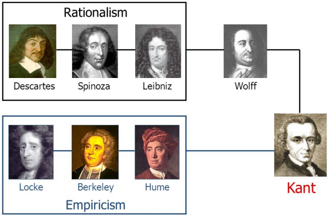
- classes, relationships between them, and constraints that hold between/for them, with possibly individuals and their relations
- as a representation of a particular subject domain

The philosophical study of ontology, concerns the kinds of entities that exist, the nature of their existence, what kinds of properties they have, and how they may be identified and categorised: such questions as are objects infinitely divisible? are causation and correlation the same? what defines a human being?

It includes *mereology* (the study of parthood), and representations of temporal, spatial, and epistemic information.

Pragmatically, it consists of taxonomies, database schemas, and knowledge base axioms required to represent information.

Historical Overview



Why Ontologies?

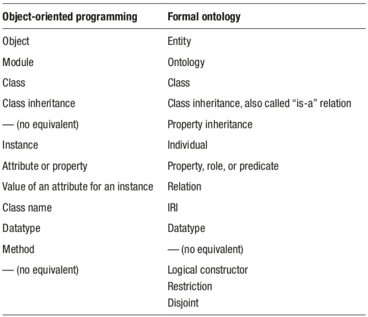
Concretely, formal ontologies can be used to achieve two objectives:

- **Perform automatic reasoning**: Formal ontologies allow logical deductions to be made, using a reasoner. For example, an ontology of animals can deduce that a white and black striped animal is actually a zebra. p

- **Link knowledge from different sources**: Formal ontologies use Internet addresses (called IRI, Internationalized Resource Identifier) to identify different entities. Any ontology can refer to any other and you can *define equivalence relationships* between existing entities when the same concept is defined twice.

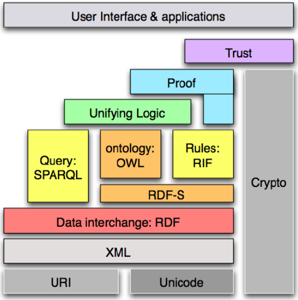
Ontologies Representations

- Several formalisms, but the web ontology language *OWL* most popular.
- Protege is a graphical OWL editor.
- OwlReady2 is a python library for OWL.
- Resource Description Framework (RDF) gives a structural description of the ontology.



Object-oriented programming	Formal ontology
Object	Entity
Module	Ontology
Class	Class
Class inheritance	Class inheritance, also called "is-a" relation
— (no equivalent)	Property inheritance
Instance	Individual
Attribute or property	Property, role, or predicate
Value of an attribute for an instance	Relation
Class name	IRI
Datatype	Datatype
Method	— (no equivalent)
— (no equivalent)	Logical constructor Restriction Disjoint

Ontologies on the Web: the layer cake



- XML - Surface syntax, no semantics
- XML Schema - Describes structure of XML documents
- RDF - Datamodel for "relations" between "things"
- RDF Schema - RDF Semantics (Mereology)
- SHACL - RDF Semantics (Validation)
- OWL - RDF Semantics (even more expressive).

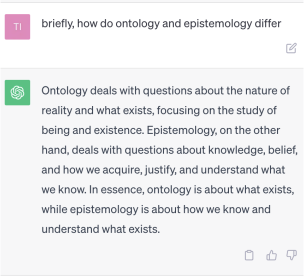
Deductive Knowledge

As humans, we can deduce more from a knowledge graph than what the edges explicitly indicate.

Given the data as premises, and some rules about the world that we may know a priori, we can use a *deductive process* to derive new data,

These types of general premises and rules, when shared by many people, form part of *commonsense knowledge* and shared by a few experts in an area, they form part of *domain knowledge*.

Machines do not have a priori access to such deductive faculties; they need to be given formal instructions, in terms of premises and entailment regimes to make similar deductions.

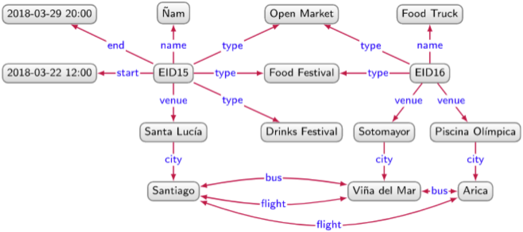


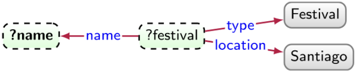
briefly, how do ontology and epistemology differ

Ontology deals with questions about the nature of reality and what exists, focusing on the study of being and existence. Epistemology, on the other hand, deals with questions about knowledge, belief, and how we acquire, justify, and understand what we know. In essence, ontology is about what exists, while epistemology is about how we know and understand what exists.

Example

Assume we are interested in knowing the festivals located in Santiago. The graph pattern returns no results for the graph as there is no node named Festival, and nothing has the location Santiago.
However, an answer (Ñam) could be automatically entailed were we to state that X being a Food Festival entails that X is a Festival, or that X having venue Y in city Z entails that X has location Z.



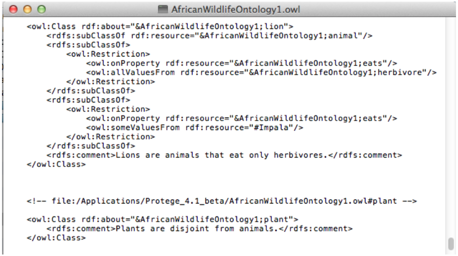


Representing Deductive Knowledge

In computing, and ontology is a concrete, formal representation of what terms mean within a given domain.

For example, one event ontology may define that if an entity is an *event*, then it has precisely one venue and precisely one time instant in which it begins. A different event ontology may define that an *event* can have multiple venues and multiple start times.

Amongst the most popular ontology languages used in practice are the Web Ontology Language (OWL), recommended by the W3C and compatible with RDF graphs; and the Open Biomedical Ontologies Format (OBOF). We will use OWL as it is more widely used.



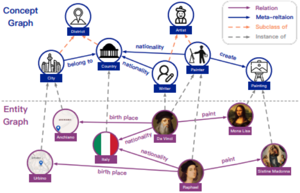
```
<owl:Class rdf:about="&AfricanWildlifeOntology1;lion">
    <rdfs:subClassOf rdf:resource="&AfricanWildlifeOntology1;animal"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&AfricanWildlifeOntology1;eats"/>
            <owl:allValuesFrom rdf:resource="&AfricanWildlifeOntology1;herbivore"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&AfricanWildlifeOntology1;eats"/>
            <owl:someValuesFrom rdf:resource="#Impala"/>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:comment>Lions are animals that eat only herbivores.</rdfs:comment>
</owl:Class>



<!-- file:/Applications/Protege_4.1_beta/AfricanWildlifeOntology1.owl#plant -->

<owl:Class rdf:about="&AfricanWildlifeOntology1;plant">
    <rdfs:comment>Plants are disjoint from animals.</rdfs:comment>
</owl:Class>
```

Interpretations and Models

We interpret the node Santiago in the data graph as referring to the real-world city that is the capital of Chile. We may further interpret an edge Arica $\longrightarrow_{\text{flight}}$ Santiago as stating that there are flights from the city of Arica to this city.

We thus interpret the data graph as another graph, the *domain graph*, composed of real-world entities connected by real-world relations. The process of interpretation, here, involves mapping the nodes and edges in the data graph to nodes and edges of the domain graph.

We can abstractly define an interpretation of a data graph as being composed of two elements: a domain graph, and a mapping from the terms of the data graph to those of the domain graph. The domain graph follows the same model as the data graph.



Definitions

A graph interpretation gives the semantics of a graph can be defined.

Definition

Graph interpretation A graph interpretation I is defined as a pair $I = (\Gamma, \cdot)$ where $\Gamma = (V_\Gamma, E_\Gamma, L_\Gamma)$ is a (directed edge-labelled) graph called the domain graph and $\cdot^I : Con \to V_\Gamma$ is a partial mapping from constants to terms in the domain graph.

We denote the domain of the mapping $\cdot^I$ by $\mathrm{dom}(\cdot^I)$. Interpretations that satisfy a graph are then said to be models of that graph.

Definition

Graph models Let $G = (V, E, L)$ be a directed edge-labelled graph. An interpretation $I = (\Gamma, \cdot^I)$ satisfies G if and only if the following hold:

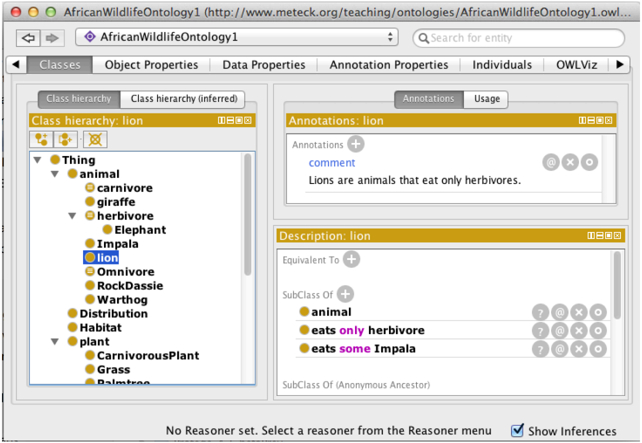
- $V \cup L \subseteq \mathrm{dom}(\cdot^I)$;
- for all $v \in V$, it holds that $v^I \in V_\Gamma$;
- for all $l \in L$, it holds that $l^I \in L_\Gamma$; and
- for all $(u, l, v) \in E$, it holds that $(u^I, l^I, v^I) \in E_\Gamma$.

If I satisfies G we call I a (graph) model of G.

Graph Model: Example

Web Ontology Language

The Web Ontology Language uses RDF as a document interchange format, so all OWL can be expressed as RDF, but OWL conforms to description logic specifications.
That is, OWL is RDF with a semantic content.



comment
Lions are animals that eat only herbivores.

SubClass Of
- animal
- eats only herbivore
- eats some Impala

No Reasoner set. Select a reasoner from the Reasoner menu

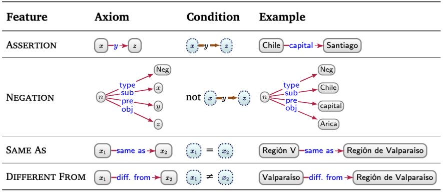
☑ Show Inferences

Ontology features: Individuals

We list the main features supported by OWL for describing individuals,

- We can assert (binary) relations between individuals using edges.
- OWL further allows for defining relations to explicitly state that two terms refer to the same entity.
- OWL can define that two terms refer to different entities.
- We may also state that a relation does not hold using negation, which can be serialised as a graph using a form of reification.

Table 4.1: Ontology features for individuals

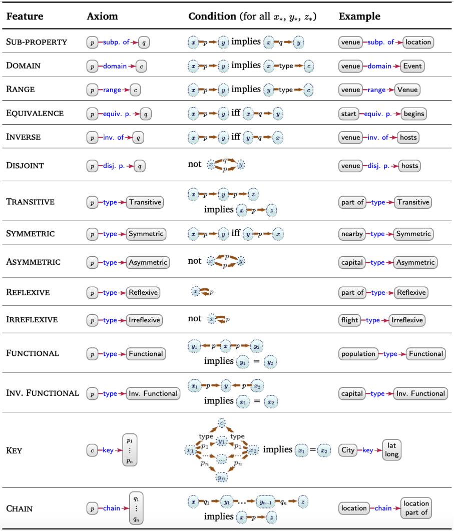


Feature	Axiom	Condition	Example
Assertion	$x \xrightarrow{y} z$	$x \xrightarrow{y} z$	Chile $\xrightarrow{\text{capital}}$ Santiago
Negation	$n \xrightarrow{\text{type}}$ Neg $n \xrightarrow{\text{sub}} x$ $n \xrightarrow{\text{pre}} y$ $n \xrightarrow{\text{obj}} z$	not $x \xrightarrow{y} z$	$n \xrightarrow{\text{type}}$ Neg $n \xrightarrow{\text{sub}}$ Chile $n \xrightarrow{\text{pre}}$ capital $n \xrightarrow{\text{obj}}$ Arica
Same As	$x_1 \xrightarrow{\text{same as}} x_2$	$x_1 = x_2$	Región V $\xrightarrow{\text{same as}}$ Región de Valparaíso
Different From	$x_1 \xrightarrow{\text{diff. from}} x_2$	$x_1 \neq x_2$	Valparaíso $\xrightarrow{\text{diff. from}}$ Región de Valparaíso

Ontology features: Properties

OWL includes RDFS type definitions, and may also define a pair of properties to be *equivalent*, *inverses*, *disjoint*, or a property to be *transitive*, *symmetric*, *asymmetric*, *reflexive*, or *irreflexive* relation.

It can can also define the multiplicity of prooperties being *functional* (many-to-one) or *inverse-functional* (one-to-many). A *key* for a class, denotes the set of properties whose values uniquely identify the entities of that class. We can relate a property to a *chain* (a path expression only allowing concatenation of properties).



Feature	**Axiom**	**Condition** (for all x_*, y_*, z_*)	**Example**
Sub-property	$p \xrightarrow{\text{subp. of}} q$	$x \xrightarrow{p} y$ implies $x \xrightarrow{q} y$	venue $\xrightarrow{\text{subp. of}}$ location
Domain	$p \xrightarrow{\text{domain}} c$	$x \xrightarrow{p} y$ implies $x \xrightarrow{\text{type}} c$	venue $\xrightarrow{\text{domain}}$ Event
Range	$p \xrightarrow{\text{range}} c$	$x \xrightarrow{p} y$ implies $y \xrightarrow{\text{type}} c$	venue $\xrightarrow{\text{range}}$ Venue
Equivalence	$p \xrightarrow{\text{equiv. p.}} q$	$x \xrightarrow{p} y$ iff $x \xrightarrow{q} y$	start $\xrightarrow{\text{equiv. p.}}$ begins
Inverse	$p \xrightarrow{\text{inv. of}} q$	$x \xrightarrow{p} y$ iff $y \xrightarrow{q} x$	venue $\xrightarrow{\text{inv. of}}$ hosts
Disjoint	$p \xrightarrow{\text{disj. p.}} q$	not $x \xrightarrow{q} y$, $x \xrightarrow{p} y$	venue $\xrightarrow{\text{disj. p.}}$ hosts
Transitive	$p \xrightarrow{\text{type}}$ Transitive	$x \xrightarrow{p} y \xrightarrow{p} z$ implies $x \xrightarrow{p} z$	part of $\xrightarrow{\text{type}}$ Transitive
Symmetric	$p \xrightarrow{\text{type}}$ Symmetric	$x \xrightarrow{p} y$ iff $y \xrightarrow{p} x$	nearby $\xrightarrow{\text{type}}$ Symmetric
Asymmetric	$p \xrightarrow{\text{type}}$ Asymmetric	not $y \xrightarrow{p} x$, $x \xrightarrow{p} y$	capital $\xrightarrow{\text{type}}$ Asymmetric
Reflexive	$p \xrightarrow{\text{type}}$ Reflexive	$x \xrightarrow{p} x$	part of $\xrightarrow{\text{type}}$ Reflexive
Irreflexive	$p \xrightarrow{\text{type}}$ Irreflexive	not $x \xrightarrow{p} x$	flight $\xrightarrow{\text{type}}$ Irreflexive
Functional	$p \xrightarrow{\text{type}}$ Functional	$y_1 \xleftarrow{p} x \xrightarrow{p} y_2$ implies $y_1 = y_2$	population $\xrightarrow{\text{type}}$ Functional
Inv. Functional	$p \xrightarrow{\text{type}}$ Inv. Functional	$x_1 \xrightarrow{p} y \xleftarrow{p} x_2$ implies $x_1 = x_2$	capital $\xrightarrow{\text{type}}$ Inv. Functional
Key	$c \xrightarrow{\text{key}} (p_1, \vdots, p_n)$	$x_1 \xrightarrow{\text{type}} c \xleftarrow{\text{type}} x_2$, $x_1 \xrightarrow{p_1} y_1 \xleftarrow{p_1} x_2$, $x_1 \xrightarrow{\dots} \dots \xleftarrow{\dots} x_2$, $x_1 \xrightarrow{p_n} y_n \xleftarrow{p_n} x_2$ implies $x_1 = x_2$	City $\xrightarrow{\text{key}}$ (lat, long)
Chain	$p \xrightarrow{\text{chain}} (q_1, \vdots, q_n)$	$x \xrightarrow{q_1} y_1 \dots y_{n-1} \xrightarrow{q_n} z$ implies $x \xrightarrow{p} z$	location $\xrightarrow{\text{chain}}$ (location, part of)

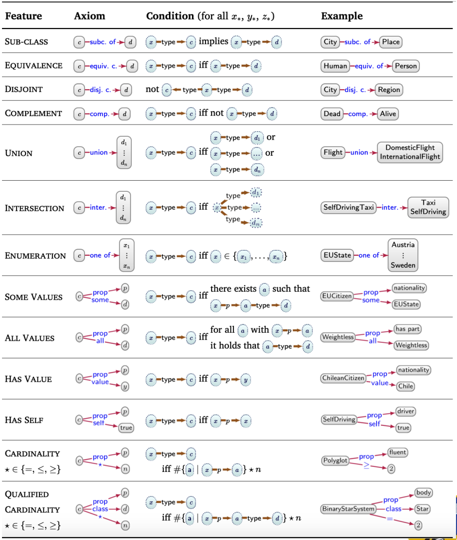
Ontology features: Classes

OWL supports RDFS class properties, and OWL can define classes to be *equivalent*, or *disjoint*.

OWL provides features for defining novel classes by applying set operators on other classes, or based on conditions that the properties of its instances satisfy:

OWL can define a novel class as the *complement* of another class, the *union* or *intersection* of a list of other classes, or an *enumeration* of all of its instances.

OWL can define classes whose instances are *some value* from a given class; *all values* from a given class; have a specific *value*; have the *self* as a reflexive value; have *at least*, *at most* or *exactly* some number of values, or values of a given class.



Feature	**Axiom**	**Condition** (for all x_*, y_*, z_*)	**Example**
SUB-CLASS	$c \xrightarrow{\text{subc. of}} d$	$x \xrightarrow{\text{type}} c$ implies $x \xrightarrow{\text{type}} d$	City $\xrightarrow{\text{subc. of}}$ Place
EQUIVALENCE	$c \xrightarrow{\text{equiv. c.}} d$	$x \xrightarrow{\text{type}} c$ iff $x \xrightarrow{\text{type}} d$	Human $\xrightarrow{\text{equiv. of}}$ Person
DISJOINT	$c \xrightarrow{\text{disj. c.}} d$	not $c \xleftarrow{\text{type}} x \xrightarrow{\text{type}} d$	City $\xrightarrow{\text{disj. c.}}$ Region
COMPLEMENT	$c \xrightarrow{\text{comp.}} d$	$x \xrightarrow{\text{type}} c$ iff not $x \xrightarrow{\text{type}} d$	Dead $\xrightarrow{\text{comp.}}$ Alive
UNION	$c \xrightarrow{\text{union}} d_1 \; \vdots \; d_n$	$x \xrightarrow{\text{type}} c$ iff $x \xrightarrow{\text{type}} d_1$ or $x \xrightarrow{\text{type}} \dots$ or $x \xrightarrow{\text{type}} d_n$	Flight $\xrightarrow{\text{union}}$ DomesticFlight InternationalFlight
INTERSECTION	$c \xrightarrow{\text{inter.}} d_1 \; \vdots \; d_n$	$x \xrightarrow{\text{type}} c$ iff $x \xrightarrow{\text{type}} d_1$ $x \xrightarrow{\text{type}} \dots$ $x \xrightarrow{\text{type}} d_n$	SelfDrivingTaxi $\xrightarrow{\text{inter.}}$ Taxi SelfDriving
ENUMERATION	$c \xrightarrow{\text{one of}} x_1 \; \vdots \; x_n$	$x \xrightarrow{\text{type}} c$ iff $x \in \{x_1, \dots, x_n\}$	EUState $\xrightarrow{\text{one of}}$ Austria ⋮ Sweden
SOME VALUES	$c \xrightarrow{\text{prop}} p$ $c \xrightarrow{\text{some}} d$	$x \xrightarrow{\text{type}} c$ iff there exists a such that $x \xrightarrow{p} a \xrightarrow{\text{type}} d$	EUCitizen $\xrightarrow{\text{prop}}$ nationality EUCitizen $\xrightarrow{\text{some}}$ EUState
ALL VALUES	$c \xrightarrow{\text{prop}} p$ $c \xrightarrow{\text{all}} d$	$x \xrightarrow{\text{type}} c$ iff for all a with $x \xrightarrow{p} a$ it holds that $a \xrightarrow{\text{type}} d$	Weightless $\xrightarrow{\text{prop}}$ has part Weightless $\xrightarrow{\text{all}}$ Weightless
HAS VALUE	$c \xrightarrow{\text{prop}} p$ $c \xrightarrow{\text{value}} y$	$x \xrightarrow{\text{type}} c$ iff $x \xrightarrow{p} y$	ChileanCitizen $\xrightarrow{\text{prop}}$ nationality ChileanCitizen $\xrightarrow{\text{value}}$ Chile
HAS SELF	$c \xrightarrow{\text{prop}} p$ $c \xrightarrow{\text{self}}$ true	$x \xrightarrow{\text{type}} c$ iff $x \xrightarrow{p} x$	SelfDriving $\xrightarrow{\text{prop}}$ driver SelfDriving $\xrightarrow{\text{self}}$ true
CARDINALITY $\star \in \{=, \leq, \geq\}$	$c \xrightarrow{\text{prop}} p$ $c \xrightarrow{\star} n$	$x \xrightarrow{\text{type}} c$ iff $\#\{a \mid x \xrightarrow{p} a\} \star n$	Polyglot $\xrightarrow{\text{prop}}$ fluent Polyglot $\xrightarrow{\geq}$ 2
QUALIFIED CARDINALITY $\star \in \{=, \leq, \geq\}$	$c \xrightarrow{\text{prop}} p$ $c \xrightarrow{\text{class}} d$ $c \xrightarrow{\star} n$	$x \xrightarrow{\text{type}} c$ iff $\#\{a \mid x \xrightarrow{p} a \xrightarrow{\text{type}} d\} \star n$	BinaryStarSystem $\xrightarrow{\text{prop}}$ body BinaryStarSystem $\xrightarrow{\text{class}}$ Star BinaryStarSystem $\xrightarrow{=}$ 2

More Axioms: SWRL

SWRL allows general axioms to be specified in OWL documents.
SWRL (Semantic Web Rule Language) is a language that allows you to integrate inference rules into ontologies. Rules can be written in the Protégé editor or in Python, using Owlready, and then executed via the integrated HermiT or Pellet reasoners.
A SWRL rule includes one or more conditions and one or more consequences, separated by an arrow "->" (composed of the two characters: minus and greater than). If the rule has several conditions or consequences, they are separated from each other by a comma "," (and)
The elements that make up conditions and consequences are called atoms, and the same atoms can be used in conditions and in consequences.

Semantic Conditions

We define models under semantics conditions.

Definition

Semantic condition Let 2^G denote the set of all (directed edge-labelled) graphs. A semantic condition is a mapping $\phi : 2^G \rightarrow \{true, false\}$. An interpretation $I := (\Gamma, \cdot^I)$ is *a model of G under ϕ* if and only if $I$is a model of G and $\phi(\Gamma)$ is true. Given a set of semantic conditions Φ, we say that I is a model of G if and only if I is a model of G and for all $\phi \in \Phi$, $\phi(\Gamma)$ is true.

For example, we can define the *Has Value* semantic condition as:

$$\forall c, p, y((\Gamma(c, prop^I, p) \land \Gamma(c, value^I, y)) \leftrightarrow \forall x(\Gamma(x, type^I, c) \rightarrow (x, p, y)))$$

Here we overload Γ as a ternary predicate to capture the edges of Γ.

Entailment

Like RDFS, the conditions listed in the previous tables give rise to entailments.

We say that one graph entails another if and only if any model of the former graph is also a model of the latter graph. Intuitively this means that the latter graph says nothing new over the former graph and thus holds as a logical consequence of the former graph.

Definition

Graph entailment Letting G_1 and G_2 denote two (directed edge-labelled) graphs, and Φ a set of semantic conditions, we say that G_1 entails G_2 under Φ – denoted $G_1 \models_{\Phi} G_2$ – if and only if any model of G_1 under Φ is also a model of G_2 under Φ.

OWLReady 2

The get_ontology() function allows you to create an empty ontology from its IRI (it is preferable to indicate the separator, “#” or “/”, at the end of the IRI, because Owlready cannot guess it since the ontology is empty!):

```
>>> from owlready2 import *
>>> onto = get_ontology("http://test.org/onto.owl#")
```

Owlready2 includes RDFLib, and you can access the triples through its default_world object:

```
>>> graph = default_world.as_rdflib_graph()
```

See https://owlready2.readthedocs.io/en/latest/world.html for more details.

Defining Classes

To create an OWL class, simply create a Python class that inherits from Thing. For example, we can create the Bacterium, Shape, and Grouping classes as follows:

```
>>> with onto:
...     class Bacterium(Thing): pass
...     class Shape(Thing): pass
...     class Grouping(Thing): pass
```

OWL properties are actually "classes of relationship". Properties inherit from DataProperty, ObjectProperty, or AnnotationProperty or others.

```
>>> with onto:
...     class has_shape(ObjectProperty, FunctionalProperty):
...         domain = [Bacterium]
...         range  = [Shape]
```

Creating Individuals

Individuals are created like any other instance in Python, by callin the class:

```
>>> my_bacterium = Bacterium()
```

Owlready automatically assigns a new IRI to the individual, created by taking the IRI of the ontology and adding the name of the class in lowercase followed by a number starting at 1:

```
>>> my_bacterium.iri
’http://test.org/onto.owl#bacterium1’

>>> my_bacterium is onto.bacterium1
True
```

Modifying Properties

Relationships between individuals and existential restrictions can be modified like any other attribute in Python. For example, it is possible to modify an individual's relationships as follows:

```
>>> my_bacterium.gram_positive = True
If it is a property of type ObjectProperty, a new instance of th
expected class can be created (here, the Rod class):
>>> my_bacterium.has_shape = Rod()
>>> my_bacterium.has_shape
onto.rod1
```

```
>>> with onto:
...     class Bacterium(Thing): pass
...     class Shape(Thing): pass
...     class has_shape(Bacterium >> Shape): pass
...     class Bacterium(Thing):
...         has_shape = Shape
```

The save() method allows saving an ontology on disk:

```
onto.save(file)
```

Description Logic

In Owlready, restrictions are created with the syntax `property.restriction_type(value)`, using the same keywords for restriction types as in Protected:

- property.some(Class) for an existential restriction
- property.only(Class) for a universal restriction
- property.value(individual or data) for a value restriction (also called role-filler)
- property.exactly(cardinality, Class) for an exact cardinality restriction
- property.min(cardinality, Class) and property. max(cardinality, Class) for minimal and maximal cardinality restrictions, respectively

Description Logic cont.

The logical operators NOT (complement), AND (intersection), and OR (union) are obtained as follows:

- `Not(Class)`
- `And([Class1, Class2,...])` or `Class1 & Class2 & ...`
- `Or([Class1, Class2,...])` or `Class1 | Class2 | ...`
- A set of individuals is obtained as follows: `OneOf([individual1, individual2,...])`
- The inverse of a property is obtained as follows: `Inverse(Property)`
- A property chain is obtained as follows: `PropertyChain([Property1, Property2,...])`

Examples

```
with onto:
  class Pseudomonas(onto.Bacterium):
    is_a = [
      onto.has_shape.some(onto.Rod),
      onto.has_shape.only(onto.Rod),
      onto.has_grouping.some(onto.Isolated | onto.InPair),
      onto.gram_positive.value(False)
    ]

  constructor = onto.Streptococcus.equivalent_to[0]
  if isinstance(constructor, And):
    print("And", constructor.Classes)
```

prints:

```
constructor.type, constructor.value)
And [bacteria.Bacterium,
     bacteria.has_shape.some(bacteria.Round),
     bacteria.has_shape.only(bacteria.Round),
     bacteria.has_grouping.some(bacteria.InSmallChain),
     bacteria.has_grouping.only(Not(bacteria.Isolated)),
     bacteria.gram_positive.value(True)]
```

Summary

- Ontologies are used for representing abstract semantic properties of knowledge graphs.
- They have a powerful lanagueg for defining semantic conditions on knowledge graphs.
- OWL is an extension of RDF to represent these semangtic conditions
- owlready2 is a python library that extends the functionality of rdflib to the OWL format.

Next week we will look at entailment and reasoning with OWL, Description Logics and the Pellet Reasoner.