OWL

web-based knowledge representation
13. February 2009
Stefan Schlobach

Context

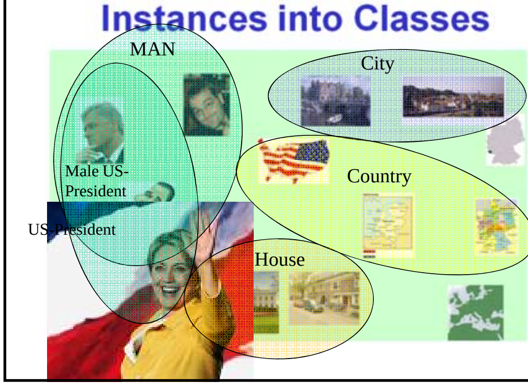
- Last week
  - Ontologies
- Wednesday
  - RDF and RDF Schema
    - simple knowledge representation for web resources
- This lecture
  - RDF Schema ++
    - more knowledge representation constructs

Ontologies: a recap

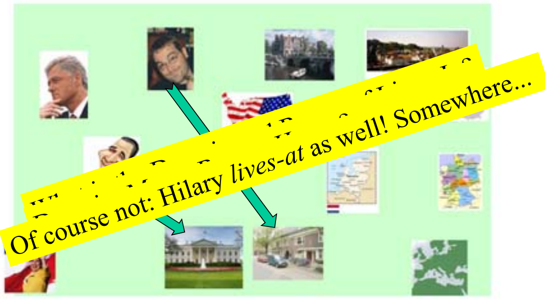
Ontologies: Some Instances in the Universe

Ontologies: Ordering Instances into Classes

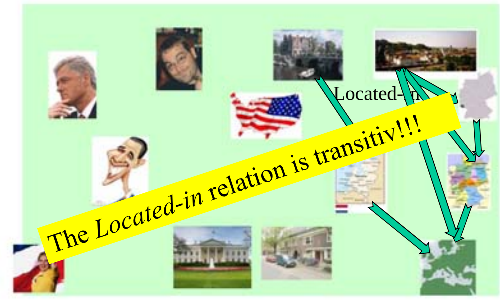
Ontologies: Ordering Instances into Classes



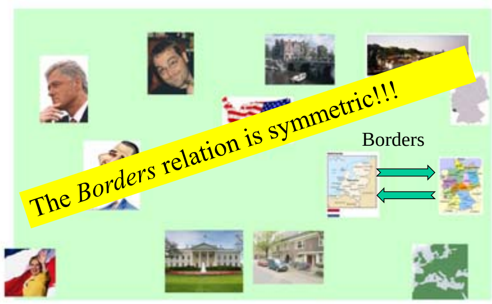
Ontologies: Properties



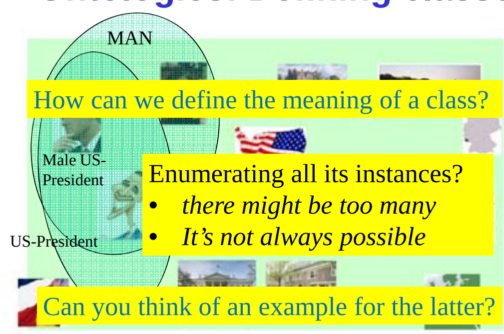
Ontologies: Properties



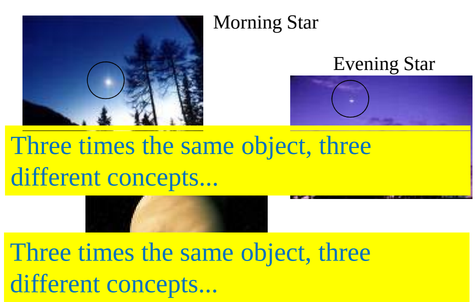
Ontologies: Properties



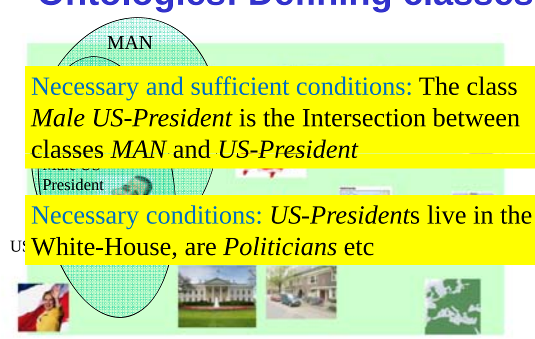
Ontologies: Defining classes



Frege's famous example



Ontologies: Defining classes



Ontologies: Restrictions

Existential restriction: a Maritime-Nation is a Country AND there exists a border with the Sea

Universal restriction: what would be a Country AND which only borders with the Sea?

An Island...

Web ontology languages

- But first, let us take a 5 minute break

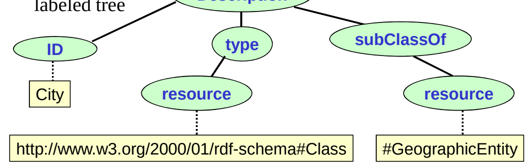
Retrospection

- Different animals:
  - XML: meta-language
  - RDF: data model
  - RDFS: ontology language
- Languages build on each other:
  - RDF: uses XML, agreement on meaning of some tags
    - examples: <Description>, type, ID
    - semantics: datamodel: subject, predicate, object
  - RDF Schema: uses RDF, agreement on meaning of some predicates
    - examples: <Class> <Property> subClassOf
    - semantics : extension of datamodel: class- and property hierarchy

What is this?

```
<rdf:Description rdf:ID="City">
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#GeographicEntity"/>
</rdf:Description>
```

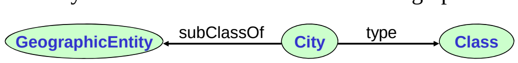
- XML
  - data model: labeled tree




```
<rdf:Description rdf:ID="City">
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#GeographicEntity"/>
</rdf:Description>
```

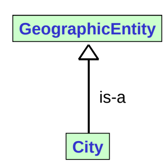
- RDF: triples
  - different data model
  - meaning of tags Description, ID and resource is interpreted

	subject	predicate	object
1.	'City'	rdf:type	#Class
2.	'City'	rdfs:subClassOf	#GeographicEntity




```
<rdf:Description rdf:ID="City">
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#GeographicEntity"/>
</rdf:Description>
```

- RDF Schema: class hierarchy
  - again different data model
  - meaning of type and subClassOf is interpreted

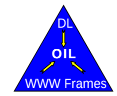


RDF(S) Limitations

- Constraints
  - Cardinality / Functionality:
    Every country has one capital
  - Disjointness: Cities cannot be US-President
- Transitivity, Inverse roles
  - SB l-in Germany, Germany l-in Europe -> SB l-in Europe
  - Lining-in versus home-of …

Ideas behind OWL

- Three roots:
  - looks like **frame based modeling languages** (like UML, Java)
  - reasoning from **Description Logic**
  - founded in **web languages** (XML, RDF)



- Goals:
  - expressive enough
  - well defined semantics
  - efficient reasoning support

Additions of OWL wrt RDFS

monarchy	republic
ruler: king	ruler: president

- Local scope of properties
  - different characteristics in different classes
- Two meanings for properties
  - **allValuesFrom** (∀): islands borders ***only*** with entities of type Sea
  - **someValuesFrom** (∃): every city consist_of at least one house
- Cardinality of properties
  - a insect has_leg **min-cardinality** 6

Additions of OWL wrt RDFS - 2

- Characteristics of properties
  - **inverse** property: define the inverse relation (e.g. has_owner is inverse of owned_by)
  - **symmetric**: if A has-prop B, then also B has-prop A (e.g. is_married_with, borders_with)
  - **transitive**: if A has-prop B and B has-prop C, then also A has-prop C. (e.g. bigger_than, located_in)
  - **functional**: maximal one value per instance (e.g. has_mother, capital)
  - **inverse-functional**: unambiguous, the only possible value (e.g. is_mother_of)

Additions of OWL wrt RDFS - 3

- Boolean expression of classes
  - **disjunction**: car or bike
  - **conjunction**: vehicle and status_symbol
  - **negation**: not animal
- Defined classes
  - not only **necessary** conditions: every lion eats meat
    - lion ⇒ eats meat
  - but also **sufficient** conditions: every person that eats fish nor meat is a vegetarian!
    - vegetarian ⇔ person eats (not (meat or fish))

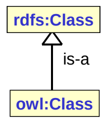
Additions of OWL wrt RDFS - 4

- Equivalence and difference
  - same **class** or **property**: car = automobile, or has_leader = has_head
  - same **individual**: VU = Vrije Universiteit
  - different **individuals**: VU ≠ UvA
- Data types
  - use **XML Schema** for data types (builtin + new)

```
<xsd:SimpleType name="lessThan200">
  <xsd:restriction base="xsd:PositiveInteger">
    <xsd:maxExclusive value="200">
  </xsd:restriction>
</rdf:Description>
```

OWL Syntax(es)

- OWL is extension of RDFS
  - same trick as before:
    - give meaning to some predicates
  - but: semantics should be compatible
- OWL builds on top of RDF-S
  - ❶ OWL is defined as **RDFS extension**
    - extension = addition to meta-model
    - new constructs besides rdfs:Class etc.
  - ❷ OWL primitives are related to RDFS
    - owl:Class ⊆ rdfs:Class



OWL Syntax(es)

- How are these extensions expressed?
- Cha
  - tw rdf:Property:
  - other characteristics are subclasses again

XML syntax:

```
<owl:FunctionalProperty rdf:ID="husband_of">
  <rdfs:domain rdf:resource="#Woman"/>
  <rdfs:range rdf:resource="#Man"/>
</owl:FunctionalProperty>
```

RDF syntax:

```
husband_of
        a owl:FunctionalProperty ;
        rdfs:domain  Woman ;
        rdfs:range Man .
```

OWL Syntax - 3

- Equivalence, disjointness, boolean expressions of classes:
  - via specific properties on rdfs:Class or rdf:Property (like rdf:subclassOf)
    - boolean expressions: owl:unionOf, owl:complementOf, owl:intersectionOf
    - element

element	equivalance	disjointness
class	owl:equivalentClass	owl:disjointWith
property	owl:equivalentProperty	-
individual	owl:sameAs	owl:differentFrom

Man owl:disjointWith Woman

OWL Syntax - 4

- Local characteristics of properties
  - via anonymous RDFS class owl:Restriction
  - the restriction class has properties:
    - owl:onProperty, and
    - owl:cardinality or owl:allValuesFrom or owl:someValuesFrom or owl:hasValue
  - class is made *subclass* of restriction

```
<owl:Class rdf:ID="Human">
<rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#has-parent"/>
      <owl:allValuesFrom rdf:resource="#Human"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

OWL Syntax - 5

- Classes can be defined by enumeration
  - via owl:oneOf
  - also **lists** can be defined
- Each ontology starts with header
  - version information
  - imports
  - comments
  - compatibility

```
<owl:Ontology rdf:about="">
   <rdfs:comment> … </rdfs:comment>
   <owl:VersionInfo> … </owl:VersionInfo>
   <owl:imports rdf:resource="http://.."/>
   <owl:priorVersion rdf:resource="http://.."/>
</owl:Ontology>
```

A formal, logical, underpinning

- But first, let us take a 10 minute break

OWL has formal semantics

- Meaning beyond words!
- Defined by mapping to very expressive Description Logic (DL)
  - eats value (meat or fish)
  - $= \exists$ eats:meat $\cup \exists$ eats:fish
- Mapping is used to provide reasoning support from a DL system (e.g., FaCT)

Benefits of formal semantics

- **Reason** about class membership, equivalence and inconsistency
  - herbivore $\Leftrightarrow$ animal eats (plant or (part_of plant))
  - tree $\Rightarrow$ plant
  - branch $\Rightarrow$ part_of tree
  - leaf $\Rightarrow$ part_of branch
  - giraffe $\Rightarrow$ animal eats leaf
  - part_of = transitive
- now we can **derive** that:
  - giraffe $\Rightarrow$ herbivore

Description Logics

- What is a logic?
  - Language (maybe different Syntaxes)
  - Meaning (Semantics)
  - Reasoning (Calculus)
- What are Description Logics
  - Set Description Languages
  - Concepts are interpreted as Sets
  - Logical Reasoning is supported

A crash course in (D)L

- Syntax: recursively define a language
  - Atomic classes and relations: e.g. Man, Father
  - Recursively define all possible classes:
    - C is a concept implies that (not C) is a concept
    - C,D concepts implies that C v D is a concept.
    - R relation, C concept -> (all r C) is a concept.
  - What can we say about classes and instances?
    - C is a D
    - i:C and (i,j):R

A crash course in (D)L

- Formal semantics: a well understood structure, and a relation to the language
- An interpretation I = (U,I(.))
  - Atomic names are interpreted as subsets of U
  - Relation symbols are interpreted as relations
    - I(C v D) = I(C) u I(D)
    - I(all r C) = {x| all y: r(x,y) -> y:I(C)}
- We relate our DL language to well-known set theory…

A crash course in (D)L

- Reasoning is now based on these semantics
  - A class C is satisfiable if there is an interpretation I such that I(C) not empty.
  - Concept (and Dog (not Dog)) is unsatisfiable, because I(and Dog (not Dog))
    - = I(Dog) intersect I(not Dog)
    - = I(Dog) intersect U \ I(Dog) = empty
- Calculi to decide for every DL formula, whether a class is satisfiable or not.

What does this buy me?

- DL reasoning is domain independent
- DL reasoning is application independent
- DL reasoning is optimized (once and for all)
- DL reasoning is well understood (complexity)
- OWL(DL) can be translated into DL
  - therefore, DL reasoning can be used for OWL
  - but DL are general purpose KR languages

Why reasoning support?

- Important
  - as **design support** tool
  - for large ontologies
  - with multiple authors
  - for **integrating and sharing** ontologies
- because it allows to
  - Establish inter-ontology relationships
  - Check for **consistency**
  - Check for (unexpected) implied relationships
- Shown useful for DB schema integration
- Can facilitate query answering

OWL Layered approach

- One size doesn't fit all; complaints:
  - "some constructs are difficult to understand"
    - *inverse functional, negation, disjunction*
  - "language should be decidable"
  - "more expressiveness"
- Solution:
  - layered approach with extended capabilities, e.g.
    - simple, intuitive modeling, or
    - efficient reasoning, or
    - high expressiveness, or ...

Putting it all together

Putting it all together

- What can we do with these things?
- General picture:
  - Semantic web requires:
    - structured **data**
    - **knowledge** of where the data is about (ontology)
- What about
  - XML on itself:
    - provides structured data in documents
    - **no** knowledge of where the data is about
    - ⇒ not appropriate for reasoning on the web

Putting it all together - 2

- RDF(S) techniques:
  - **data** is captured in RDF descriptions

```
<rdf:Description ID="Z31">
  <rdf:type resource="http://my.business.com/prod#Inkjet"/>
  <onto:price>199,-</onto:price>
</rdf:Description>
```

  - some **knowledge** is described in RDF Schema

```
<rdfs:Class ID="Inkjet">
  <rdfs:subClassOf resource="#Printer"/>
</rdfs:Class>
```

  ⇒ possible to find out that "Z31" is a printer of type inkjet with price 199,-

Putting it all together - 3

– extended techniques (OWL):
  - **data** is still captured in RDF descriptions
  - **knowledge** is captured in OWL ontology

```
<owl:Class rdf:ID="CheapPrinter">
   <owl:intersectionOf>
      <rdfs:Class rdf:about="#Printer"/>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#price"/>
         <owl:hasClass rdf:resource="...#lessThan200"/>
      </owl:Restriction>
   </owl:intersectionOf>
</owl:Class>
```

⇒ possible to find “Z31” when looking for a CheapPrinter

⇒ check whether Inkjet and CheapPrinter are equivalent, etc.

OWL as a Web language

```
dbpedia:Saarbruecken geonames:located-In dbpedia:Saarland .

dbpedia:Saarland located-In sumo:Germany .

sumo:Germany located-In cyc:Europe .

located-In owl:transitiveProperty .
```

```
SPARQL: RDF query language:
SELECT ?location WHERE
        { dbpedia:Saarbruecken located-In ?location }
```

Results:
- With OWL: dbpedia:Saarland, sumo:Germany, cyc: Europe
- Without OWL: dbpedia:Saarland

Assignment

- Develop a small ontology with Protégé:
  - define a class via “necessary and sufficient” conditions
  - define another class that is *implicitly* a subclass of the defined class
  - show that the tool classifies the other class correctly

  (similar to giraffe example)