University of Southampton

Web Ontology Language (OWL)

COMP6256 Knowledge Graphs for AI Systems

Dr Nicholas Gibbins – nmg@ecs.soton.ac.uk

University of Southampton

Introducing OWL

For many, RDF Schema is a sufficiently expressive ontology language

However, there are use cases which require a more expressive formalism:

- Instance classification
- Consistency checking
- Subsumption reasoning

OWL is a way of encoding DL axioms as RDF triples such that the semantics of the DL axioms are broadly compatible with RDF(S)

OWL Feature Summary

- Necessary and sufficient conditions for class membership
- Property restrictions
  - Local range, cardinality, value constraints
- Equivalence and identity relations
- Property characteristics
  - Transitive, symmetric, functional
- Complex classes
  - Set operators, enumerated classes, disjoint classes

OWL Versions

Two versions of OWL:

- OWL 1.0 (became Recommendation on 10 Feb 2004)
- OWL 2 (became Recommendation on 29 Oct 2009)

OWL 2 is more expressive than OWL 1.0, and takes advantage of developments in DL reasoning techniques in the intervening time

We will initially concentrate on OWL 1.0

OWL 1.0 Species

Different subsets of OWL features give rise to the following sublanguages (colloquially known as species):

- OWL Lite
- OWL DL
- OWL Full

“There is a tradeoff between the expressiveness of a representation language and the difficulty of reasoning over the representations built using that language.”

Brachman, R. J., and H. J. Levesque. (1984). The tractability of subsumption in frame-based description languages. In Proceedings of the 4th National Conference of the American Association for Artificial Intelligence (AAAI-84). Austin, TX, pp. 34-37.

OWL 1.0 Species

University of Southampton

Complexity of reasoning in Description Logics

Note: the information here is (always) incomplete and updated often

Base description logic: $\mathcal{A}$ttributive $\mathcal{L}$anguage with $\mathcal{C}$omplements

$\mathcal{ALC}$ ::= $\bot$ | $\top$ | A | $\neg C$ | $C \sqcap D$ | $C \sqcup D$ | $\exists R.C$ | $\forall R.C$

Concept constructors:

- $\mathcal{F}$ – functionality[2]: $(\leq 1\ R)$
- $\mathcal{N}$ – (unqualified) number restrictions: $(\geq n\ R)$, $(\leq n\ R)$
- $\mathcal{Q}$ – qualified number restrictions: $(\geq n\ R.C)$, $(\leq n\ R.C)$
- $\mathcal{O}$ – nominals: $\{a\}$ or $\{a_1, \ldots, a_n\}$ ("one-of")
- μ – least fixpoint operator: $\mu X.C$
- Forbid complex roles[5] in number restrictions[6]

Role constructors: trans reg

- $\mathcal{I}$ – role inverse: R^-
- $\sqcap$ – role intersection[3]: $R \sqcap S$
- $\sqcup$ – role union: $R \sqcup S$
- $\neg$ – role complement: $\neg R$ full
- o – role chain (composition): $R \circ S$
- * – reflexive-transitive closure[4]: R^*
- *id* – concept identity: $id(C)$

TBox (concept axioms):

- empty TBox
- acyclic TBox ($A \equiv C$, A is a concept name; no cycles)
- general TBox ($C \sqsubseteq D$, for arbitrary concepts C and D)

RBox (role axioms): OWL-Lite OWL-DL OWL 1.1

- $\mathcal{S}$ – role transitivity: $Tr(R)$
- $\mathcal{H}$ – role hierarchy: $R \sqsubseteq S$
- $\mathcal{R}$ – complex role inclusions: $R \circ S \sqsubseteq R$, $R \circ S \sqsubseteq S$
- s – some additional features (click to see them)

Reset You have selected a Description Logic: $\mathcal{ALC}$

Complexity[7] of reasoning problems[8]		
Concept satisfiability	**PSpace-complete**	• Hardness for $\mathcal{ALC}$: see [80]. • Upper bound for $\mathcal{ALCQ}$: see [12, Theorem 4.6].
ABox consistency	**PSpace-complete**	• Hardness follows from that for concept satisfiability. • Upper bound for $\mathcal{ALCQO}$: see [17, Appendix A].

http://www.cs.man.ac.uk/~ezolin/dl/

OWL Lite

Description Logic-based

- SHIF(D)
- Satisfiability is ExpTime-complete

Less complex reasoning at the expense of less expressive language

- No enumerated classes, set operators, or disjoint classes
- Restricted cardinality restrictions
  (values of 0 or 1 – required, permitted and excluded)
- No value restrictions
- equivalentClass/subClassOf cannot be applied to class expressions

OWL DL

Description Logic-based

- SHOIN(D)
- Complete and decidable
- Higher worst-case complexity than OWL Lite – NExpTime-complete

Supports all OWL constructs, with some restrictions

- Properties that take datatype values cannot be marked as inverse functional
- Classes, properties, individuals and datatype values are disjoint

OWL Full

No restrictions on use of language constructs

- All OWL DL and RDFS constructs

- Potentially undecidable

OWL 1.0 Features and Syntax

Ontology header

```
<> rdf:type owl:Ontology ;
   owl:versionInfo "1.4" ;
   rdfs:comment "An example ontology" ;
   owl:imports <http://example.org/base/> .
```

`owl:versionInfo` – ontology version number, etc

`owl:priorVersion` – specified ontology is a previous version of this on

`owl:backwardCompatibleWith` – specified ontology is a previous version of this one, and that this is compatible with it

`owl:incompatibleWith` –specified ontology is a previous version of this one, but that this is incompatible with it

OWL class types

`owl:Class`

- Distinct from `rdfs:Class` – needed for OWL Lite/DL

`owl:Thing` (⊤)

- The class that includes everything

`owl:Nothing` (⊥)

- The empty class

`owl:DeprecatedClass`

- Used to indicated that a class is deprecated and should not be used

OWL property types

`owl:ObjectProperty`

- The class of resource-valued properties

`owl:DatatypeProperty`

- The class of literal-valued properties

`owl:AnnotationProperty`

- Used to type properties which annotate classes and properties (needed for OWL Lite/DL)
- Any triples whose predicates are typed as annotation properties are ignored by OWL reasoners

`owl:DeprecatedProperty`

- Used to indicated that a property is deprecated and should not be used

OWL versus RDF Schema

Recall that the semantics of a description logic is specified by interpretation functions which map:

- Instances to members of the domain of discourse
- Classes to subsets of the domain of discourse
- Properties to sets of pairs drawn from the domain of discourse

Reflexive definitions of RDF Schema means that some resources are treated as both classes and instances, or instances and properties

- Ambiguous semantics for these resources
- Can't tell from context whether they're instances or classes
- Can't select the appropriate interpretation function

The introduction of `owl:Class`, `owl:ObjectProperty` and `owl:DatatypeProperty` eliminates this ambiguity

OWL restrictions

Class expressions formed by constraints on properties:

- Local cardinality constraints
  $\leq n\, R$, $\geq n\, R$, $= n\, R$
- Local range constraints
  $\exists R.C$, $\forall R.C$
- Local value constraints
  $\exists R.\{x\}$

Common triple format for restrictions

```
[ rdf:type owl:Restriction ;
  owl:onProperty <property URI> ;
  constraint expression ]
```

Local cardinality constraints

Defines a class based on the number of values taken by a property

`owl:minCardinality` $(\geq n\, R)$

- "property R has at least n values"

`owl:maxCardinality` $(\leq n\, R)$

- "property R has at most n values"

`owl:cardinality` $(= n\, R)$

- "property R has exactly n values"

OWL Lite has restricted cardinalities – $n \in \{0,1\}$

Example: Local cardinality constraint

Single malt whiskies are whiskies which are distilled by one and only one thing

$$SingleMaltWhisky \equiv Whisky \sqcap =1.distilledBy$$

```
ont:SingleMaltWhisky rdf:type owl:Class ;
  owl:equivalentClass [ rdf:type owl:Class ;
                        owl:intersectionOf
                          ( ont:Whisky
                            [ rdf:type owl:Restriction ;
                              owl:onProperty ont:distilledBy ;
                              owl:cardinality 1 ] ) ] .
```

Local range constraints

Defines a class based on the type of property values

Distinct from global range constraint (`rdfs:range`) in RDF Schema

`owl:someValuesFrom` ($\exists R.C$)

- “there exists a value for property R of type C”

`owl:allValuesFrom` ($\forall R.C$)

- “property R has only values of type C”

Can only be used with named classes or datatypes in OWL Lite

Example: Existential restriction

Carnivores are things which eat some things which are animals

$$\text{Carnivore} \equiv \exists \text{eats}.\text{Animal}$$

```
ont:Carnivore rdf:type owl:Class ;
              owl:equivalentClass [ rdf:type owl:Restriction ;
                                    owl:onProperty ont:eats ;
                                    owl:someValuesFrom ont:Animal ] .
```

Example: Universal restriction

Vegetarians are things which eat only things which are plants

$$\text{Vegetarian} \equiv \forall \text{eats}.\,\text{Plant}$$

```
ont:Vegetarian rdf:type owl:Class ;
               owl:equivalentClass [ rdf:type owl:Restriction ;
                                     owl:onProperty ont:eats ;
                                     owl:allValuesFrom ont:Plant ] .
```

Local value constraints

Defines a class based on the existence of a particular property value

`owl:hasValue` $(\exists R.\{x\})$

- "property R has a value which is X"

Cannot be used in OWL Lite

Example: Local value constraint

Green things are things which are coloured green

$$\text{GreenThing} \equiv \exists \text{hasColour}.\{\text{green}\}$$

```
ont:GreenThing rdf:type owl:Class ;
               owl:equivalentClass [ rdf:type owl:Restriction ;
                                     owl:onProperty ont:hasColour ;
                                     owl:hasValue ont:green ] .
```

Set constructors

`owl:intersectionOf` ($C \sqcap D$)

`owl:unionOf` ($C \sqcup D$)

`owl:complementOf` ($\neg C$)

Restrictions on use with OWL Lite

- `owl:unionOf` and `owl:complementOf` cannot be used
- `owl:intersectionOf` can be used with named classes (not bNodes) and OWL restrictions only

Example: Set constructors

$$\text{GreenApple} \equiv \text{Apple} \sqcap \exists \text{hasColour}.\{\text{green}\}$$

```
ont:GreenApple rdf:type owl:Class ;
  owl:equivalentClass [ owl:intersectionOf
                          ( ont:Apple
                            [ rdf:type owl:Restriction ;
                              owl:onProperty ont:hasColour ;
                              owl:hasValue ont:green ] ) ] .
```

Equivalence and identity relations

Useful for ontology mapping

`owl:sameAs` (MorningStar = EveningStar)

`owl:equivalentClass` ($C \equiv D$)

`owl:equivalentProperty` ($R \equiv S$)

```
ont:morningStar rdf:type owl:Thing ;
                owl:sameAs ont:eveningStar
```

Non-equivalence relations

`owl:differentFrom`

- Can be used to specify a limited unique name assumption

```
ont:HarryCorbett rdf:type owl:Thing ;
                 owl:differentFrom ont:HarryHCorbett
```

OWL (and DLs in general) make the Open World Assumption

- Knowledge of world is incomplete
- If something cannot be proven true, then it isn't assumed to be false

Non-equivalence relations

`owl:AllDifferent` and `owl:distinctMembers`

- Used to specify a group of mutually distinct individuals

```
[ rdf:type owl:AllDifferent ;
  owl:distinctMembers ( ont:John ont:Paul ont:George ont:Ringo ) ] .
```

Necessary Class Definitions

Primitive / partial classes ($\sqsubseteq$)

"If we know that something is a X, then it must fulfill the conditions..."

Defined using `rdfs:subClassOf`:

$$\text{Person} \sqsubseteq \exists \text{hasBirthdate}.\top$$

```
ont:Person rdf:type owl:Class ;
           rdfs:subClassOf [ rdf:type owl:Restriction ;
                             owl:onProperty ont:hasBirthdate ;
                             owl:SomeValuesFrom owl:Thing ] .
```

Sufficient Class Definitions

Describes a subset of the class (⊒)

"If we know that something has this property, then it belongs to this class..."

Defined using `rdfs:subClassOf` – in the other direction

$$\text{Person} \sqsupseteq \exists \text{hasNationalInsuranceNumber}.\top$$

Necessary and Sufficient Class Definitions

Defined / complete classes (≡)

"If something fulfills the conditions..., then it is an X."

Defined using `owl:equivalentClass`:

$$\text{Student} \equiv \text{Person} \sqcap \exists \text{isEnrolledAt}.\text{University}$$

Note: It will be rather difficult, if not impossible, to give conditions that are both sufficient **and** necessary for a class that is as complex as Person

Property types - Inverse

Defines a property as the *inverse* of another property

$(R \equiv S^{-})$

```
ont:hasAuthor rdf:type owl:ObjectProperty ;
              owl:inverseOf ont:wrote .
```

Property types - Symmetric

A property R is *symmetric* if the following condition holds:

$$\forall x \forall y \, \langle x, y \rangle \in R^I \Leftrightarrow \langle y, x \rangle \in R^I$$

```
ont:hasSibling rdf:type owl:SymmetricProperty .
```

In DL notation: $R \equiv R^-$ (symmetric properties are their own inverses)

Property types – Transitive

A property R is *transitive* if the following condition holds:

$$\forall x \forall y \forall z \; \langle x, y \rangle \in R^I \wedge \langle y, z \rangle \in R^I \Rightarrow \langle x, z \rangle \in R^I$$

```
ont:hasAncestor rdf:type owl:TransitiveProperty .
```

In DL notation: $R \sqsubseteq R^+$

Property types – Functional

A property R is *functional* if the following condition holds:

$$\forall x \forall y \forall z\, \langle x, y \rangle \in R^I \wedge \langle x, z \rangle \in R^I \Rightarrow y = z$$

```
ont:hasNINumber rdf:type owl:FunctionalProperty .
```

(everyone has only one NI number)

(everyone has only one birthdate)

(but: people may have several means of identification)

Property types – Inverse Functional

A property R is *inverse functional* if the following condition holds:

$$\forall x \forall y \forall z \, \langle y, x \rangle \in R^I \wedge \langle z, x \rangle \in R^I \Rightarrow y = z$$

```
ont:hasNINumber rdf:type owl:InverseFunctionalProperty .
```

(people with the same NI number are the same person)

(but: many people have the same birthdate)

Cannot be used with `owl:DatatypeProperty` in OWL Lite/DL

Disjoint classes

`owl:disjointWith`

- members of one class cannot also be members of some specified other class

```
ont:Duck rdf:type owl:Class ;
         owl:disjointWith ont:Goose .
```

In DL notation: $\text{Duck} \sqcap \text{Goose} \equiv \bot$

- Cannot be used in OWL Lite

Enumerated classes

Defines a class as a direct enumeration of its members

- owl:oneOf ($C \equiv \{a, b, c\}$)

```
ont:Beatles rdf:type owl:Class ;
            owl:oneOf ( ont:John ont:Paul ont:George ont:Ringo ) .
```

- Cannot be used in OWL Lite

Ontology modularisation

`owl:imports` mechanism for including other ontologies

- Also possible to use terms from other ontologies without explicitly importing them
- Importing requires certain entailments, whereas simple use does not require (but also does not prevent) those entailments

Ontology modularisation example

Ontology 1 (ont1) contains:

```
BBB rdfs:subClassOf AAA .
```

Ontology-2 (ont2) contains:

```
ont2 owl:imports ont1 .
CCC rdfs:subClassOf BBB .
```

Ontology-2 **must** entail:

```
CCC rdfs:subClassOf AAA
```

Ontology modularisation example

Ontology 1 (ont1) contains:

```
BBB rdfs:subClassOf AAA .
```

Ontology-3 (ont3) contains:

```
CCC rdfs:subClassOf ont1:BBB .
```

Ontology-3 does **not necessarily** entail

```
CCC rdfs:subClassOf ont1:AAA .
```

OWL 2

From OWL 1 to OWL 2

OWL 1 design based on contemporary understanding of techniques for decidable, sound and complete reasoning in description logics

Our understanding has improved since 2004

Some things that looked intractable have been shown to be possible

From OWL 1 to OWL 2

Changes between 1 and 2 fall into the following categories:

- Syntactic sugar (making it easier to say things we could already say)
- Constructs for increased expressivity
- Datatype support
- Metamodelling
- Annotation

Syntactic Sugar: Disjoint Classes

OWL 1 lets us state that two classes are disjoint (owl:disjointWith)

OWL 2 lets us state that a set of classes are pairwise disjoint

```
[ rdf:type owl:AllDisjointClasses ;
  owl:members ( ont:Duck ont:Goose ont:Swan ) ] .
```

In DL notation:

$$\text{Duck} \sqcap \text{Goose} \equiv \bot$$
$$\text{Duck} \sqcap \text{Swan} \equiv \bot$$
$$\text{Goose} \sqcap \text{Swan} \equiv \bot$$

Syntactic Sugar: Disjoint Union

Allows us to define a class as the union of a number of other classes, all of which are pairwise disjoint

$$C \equiv C_1 \sqcup C_2 \sqcup \cdots \sqcup C_n$$
$$C_1 \sqcap C_2 \equiv \bot$$
$$C_1 \sqcap C_3 \equiv \bot$$
$$\ldots$$
$$C_{n-1} \sqcap C_n \equiv \bot$$

We'll look at this modelling pattern in a later lecture

Example: Disjoint Union

$$Monotreme \equiv Platypus \sqcup Echidna$$
$$Platypus \sqcap Echidna \equiv \bot$$

```
ont:Monotreme owl:disjointUnionOf ( ont:Platypus ont:Echidna ) .
```

Syntactic Sugar: Negative Property Assertions

OWL 1 lets us assert property values for an individual

OWL 2 lets us assert that an individual does not have a particular property value

```
[ rdf:type owl:NegativePropertyAssertion ;
  owl:sourceIndividual ont:John ;
  owl:assertionProperty ont:hasChild ;
  owl:targetIndividual ont:Susan ] .
```

New Constructs: Self Restriction

Defines a class of individuals which are related to themselves by a given property

```
[ rdf:type owl:Restriction ;
  owl:onProperty property ;
  owl:hasSelf “true”^^xsd:boolean ] .
```

In DL notation: $\exists R.\mathrm{Self}$

Example: Self Restriction

A narcissist is a person who loves themselves

$$Narcissist \equiv Person \sqcap \exists loves.Self$$

```
ont:Narcissist rdf:type owl:Class ;
  owl:equivalentClass
    [ rdf:type owl:Class ;
      owl:intersectionOf ( ont:Person
                           [ rdf:type owl:Restriction ;
                             owl:onProperty ont:loves ;
                             owl:hasSelf “true”^^xsd:boolean ] ) ] .
```

New Constructs: Qualified Cardinality

OWL 1 lets us either specify either the local range of a property, or the number of values taken by the property

OWL 2 lets us specify both together:

```
[ rdf:type owl:Restriction ;
  owl:onProperty ont:hasPart ;
  owl:onClass ont:Wheel ;
  owl:cardinality 4 ] .
```

In DL notation: $\exists_{=4}\, \mathrm{hasPart.\,Wheel}$ or $= 4\ \mathrm{hasPart.\,Wheel}$

Similar construct for datatype properties

New Constructs: Reflexive Properties

Allows us to assert that a property relates every object to itself

A property R is *reflexive* if the following condition holds:

$$\forall x \ \langle x, x \rangle \in R^I$$

```
ont:sameAgeAs rdf:type owl:ReflexiveProperty .
```

New Constructs: Irreflexive Properties

Allows us to assert that a property relates no object to itself

A property R is *irreflexive* if the following condition holds:

$$\forall x \ \langle x, x \rangle \notin R^I$$

```
ont:strictlyTallerThan rdf:type owl:IrreflexiveProperty .

ont:marriedTo rdf:type owl:IrreflexiveProperty .
```

New Constructs: Asymmetric Properties

A property R is *asymmetric* if the following condition holds:

$$\forall x \forall y \, \langle x, y \rangle \in R^I \Rightarrow \langle y, x \rangle \notin R^I$$

```
ont:strictlyTallerThan rdf:type owl:AsymmetricProperty .
```

but:

```
ont:marriedTo rdf:type owl:SymmetricProperty .
```

New Constructs: Disjoint Properties

We can state that two individuals cannot be related to each other by two different properties that have been declared disjoint

Two properties R and S are *disjoint* if the following condition holds:

$$R^I \cap S^I = \emptyset$$

```
ont:separatedFrom rdf:type owl:ObjectProperty ;
                  owl:propertyDisjointWith ont:contiguousWith .
```

Typical examples include antonymic relationships: closeTo – farFrom

New Constructs: Property Chain Inclusion

OWL 1 does not let us define a property as a composition of other properties

- Example: $\text{hasUncle} \equiv \text{hasParent} \circ \text{hasBrother}$

OWL 2 lets us define such property compositions:

```
ont:hasUncle rdf:type owl:ObjectProperty ;
             owl:propertyChainAxiom ( ont:hasParent ont:hasBrother ) .
```

New Constructs: Keys

OWL 1 lets us define a property to be functional, so that individuals can be uniquely identified by values of that property

OWL 2 lets us define uniquely identifying keys that comprise several properties:

```
ont:Person rdf:type owl:Class ;
           owl:hasKey ( ont:hasSSN ont:hasBirthDate ) .
```

New Constructs: Datatype Restrictions

Allows us to define subsets of datatypes that constrain the range of values allowed by a datatype

For example, the datatype of integers greater than or equal to 5:

```
ont:IntGEFive rdf:type owl:Datatype ;
  owl:withRestrictions ( [ rdf:type xsd:minInclusive “5”^^xsd:integer ] ) .
```

Metamodelling: Punning

OWL 1 required the names used to identify classes, properties, individuals and datatypes to be disjoint

OWL 2 relaxes this

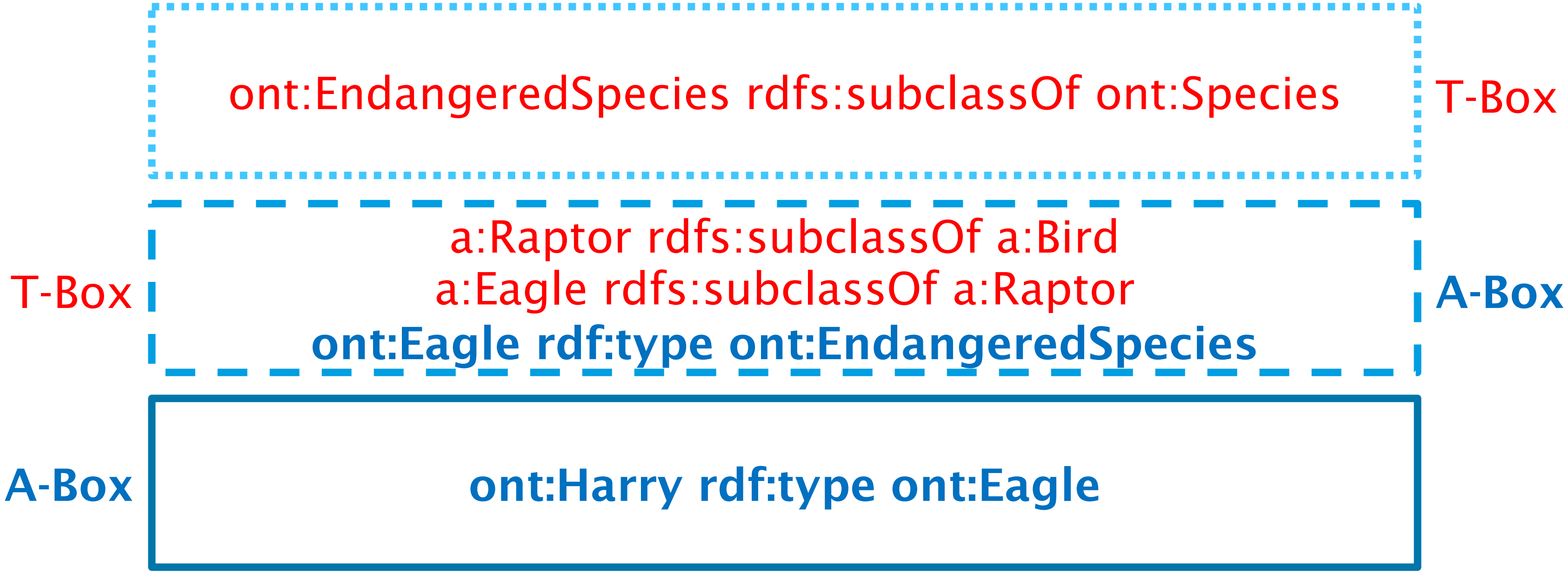
- The same name (URI) can be used for both a class and an individual

However:

- A name cannot be used for both a class and a datatype
- A name cannot be used for more than one type of property (DataProperty vs ObjectProperty)

Example: Punning

```
ont:Eagle rdf:type owl:Class .
ont:Harry rdf:type ont:Eagle .
ont:Eagle rdf:type ont:EndangeredSpecies .
```



Language Profiles

OWL 1 has three dialects: OWL Lite, OWL DL and OWL Full

OWL 2 introduces three profiles with useful computational properties (reasoning, conjunctive queries):

- OWL 2 EL (PTIME-complete, PSPACE-complete)
- OWL 2 QL (NLOGSPACE-complete, NP-complete)
- OWL 2 RL (PTIME-complete, NP-complete)

- OWL 1 DL (NEXPTIME-complete, decidability open)

Next Lecture: Ontology Engineering