Advanced OWL

Chin Zi Hau

Logics (*what a logic gives you*)

- Logic = a formal way to describe a domain using:
  - ***Axioms***: assumptions or rules we accept as true

Logic:

What *entities/relations/time* you can represent?

What kinds of *queries* or *reasoning* are possible?

    - Facts, objects, relations, time, …
  - Epistemological commitments: (*what can we know about it*)
    - True/false, possibility/necessity, belief/disbelief, certainty/uncertainty, …

Types of Logic

Simple declarative statements (proposition).

Introducing objects, properties and relationships

Uses operators like "□" (necessity) and "◊" (possibility).
What must be true? What could be true?

How things change over time.

Likelihood of events occurring.

No clear-cut boundaries, allowing partial truth.

Question: Is P(plagiarise) equivalent to ¬F(plagiarise)?

Logic languag		logical ent	Examples
Propositional		/unknown	"It is raining AND the streets are wet"
First order log		/unknown	"All cats are mammals"
Modal logic		/unknow	
Deontic logic		/unknow	**F(plagiarise)**
Temporal logi		/unknown	"I am always hungry"
Probability th		[0,1]	"There is a 70% chance of rain tomorrow"
Fuzzy logic	truth	Truth value, [0,1]	"Is it cold"? *Very Much / 0.9* *Little / 0.25* *Very Less / 0.1*

Components of a logic

Syntax

- *Rules* that specify what a well-formed sentence (or formula) looks like
- Tells us how to build a knowledge base
- All legal expressions are sentences (otherwise knows as well-formed formulas)

Semantics

- *Rules* that specify what a sentence (or formula) really means in the world
- Tells us what the knowledge base means

Propositional Logic

- The simplest type of logic
- A **proposition** is a statement that is either **true** or **false**
- There is no objects, relations, and functions.

- The symbols denote atomic statements that says something about the world
- $\neg P$ negation (means the opposite of *P*)
- $P \wedge Q$ conjunction (means both *P* and Q)
- $P \vee Q$ disjunction (means *P* or Q, or possibly both)
- $P \Rightarrow Q$ implication (means "if *P* then *Q*")
- $P \Leftrightarrow Q$ biconditional (means "*P* if and only if *Q*")

Propositional Logic vs. First Order Logic

”If Jane is younger than Lisa, then Lisa is older than Jane.”

- **Using Propositional logic:**

  p: Jane is younger than Lisa

  q: Lisa is older than Jane

  p => q

- **Using First order logic:**

  Younger: to be younger than (predicate with two variables)

  Older: to be older than

  Younger(Jane, Lisa) => **Older**(Lisa, Jane)

First Order Logic features

not Bird(Tom)

Not isMarried(Alice, Bob)

- **Atomic negation**: not C(x)
- **Role negation**: not R(x,y)

Bird ∩ Flies

Bird ∪ Mammal

- **Intersection of concepts**: $A \cap B$ (set of individuals of A v
  individuals of B)
- **Union of concepts**: $A \cup B$ (set of individuals of A and individuals of B)
- **Existential restrictions**: exists at least one individual that participates in a property (owl:some)

A person must have at least one pet that is a dog.

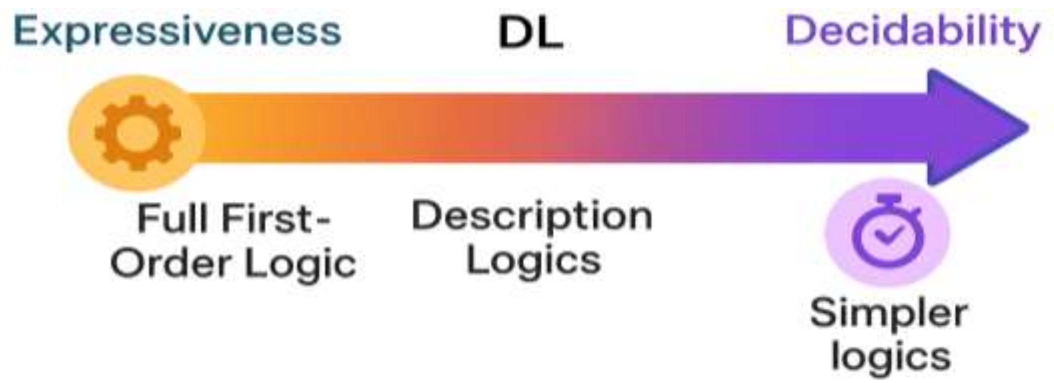
- **Universal restriction**: only (all) individuals that participate in a property (owl:only)

Only eats plant-based food.

First Order Logic syntax

- Constants: John, Richard, 2 …
- Predicates: Brother, >, Father …
- Functions: Sqrt, isKing, …
- Variables: x, y, a, b, …
- Connectives: $\neg, \Rightarrow, \Leftrightarrow, \wedge, \vee$
- Equality: =
- Quantifiers: $\forall$, $\exists$

Description Logics



- A family of formal knowledge representation languages used in AI to describe and reason about concepts.
- Description Logics strike a balance between:
  - ***Expressiveness*** – ability to *represent complex structures* and *constraints*.
  - ***Decidability*** – ability to *perform reasoning tasks* (e.g., inference) *efficiently*.
- Why not full First-Order Logic?
  - Full FOL offers ***high expressiveness but sacrifices decidability***, making reasoning computationally prohibitive.
  - DL uses selected fragments of FOL to maintain practical reasoning.

Description Logics

- In DL, three basic elements:
  - **Individuals:**
    - Specific objects (e.g., Alice, F20BD).
  - **Concepts (Classes):**
    - Groups of individuals (e.g., Person, Course).
  - **Roles (Properties):**
    - Relationships between individuals (e.g., teaches, enrolledIn).

Terms

FOL (first order logic)	DL (description logic)	OWL
constant	individual	individual
unary predicate	concept	class
binary predicate	role	property

Different Description Languages

Basic DL languages

language	name	allows
AL	Attributive language	• Atomic negation • intersection • restrictions • limited existential quantification
FL	Frame based	• Concept intersection • restrictions • limited existential quantification • role restriction
EL	Existential language	• concept intersectio • existential restrictions

Extensions

	name	
F	Functional prop	uniqueness
E	Existential qualification	There exists at least one individual
U	Concept union	
	hy	Rdfs:subPropertyOf
C	Role negation	Not p(x,y)
R	Limited role inclusion	Reflexivity, disjointness
		P(x,y) => Q(y,x)
		owl:cardinality, counting
Q	Cardinality restriction	(in owl2)
(D)	Use of datatype prop	Data values or data types

∃hasPart.Wheel

Specifying constraints on the typ
objects that can fill a role

a person can have at most 2 parents

∃hasPart.(Wheel ⊓ HasColor.Red)

S is used as abbreviation for ALC, some use it for ALCH

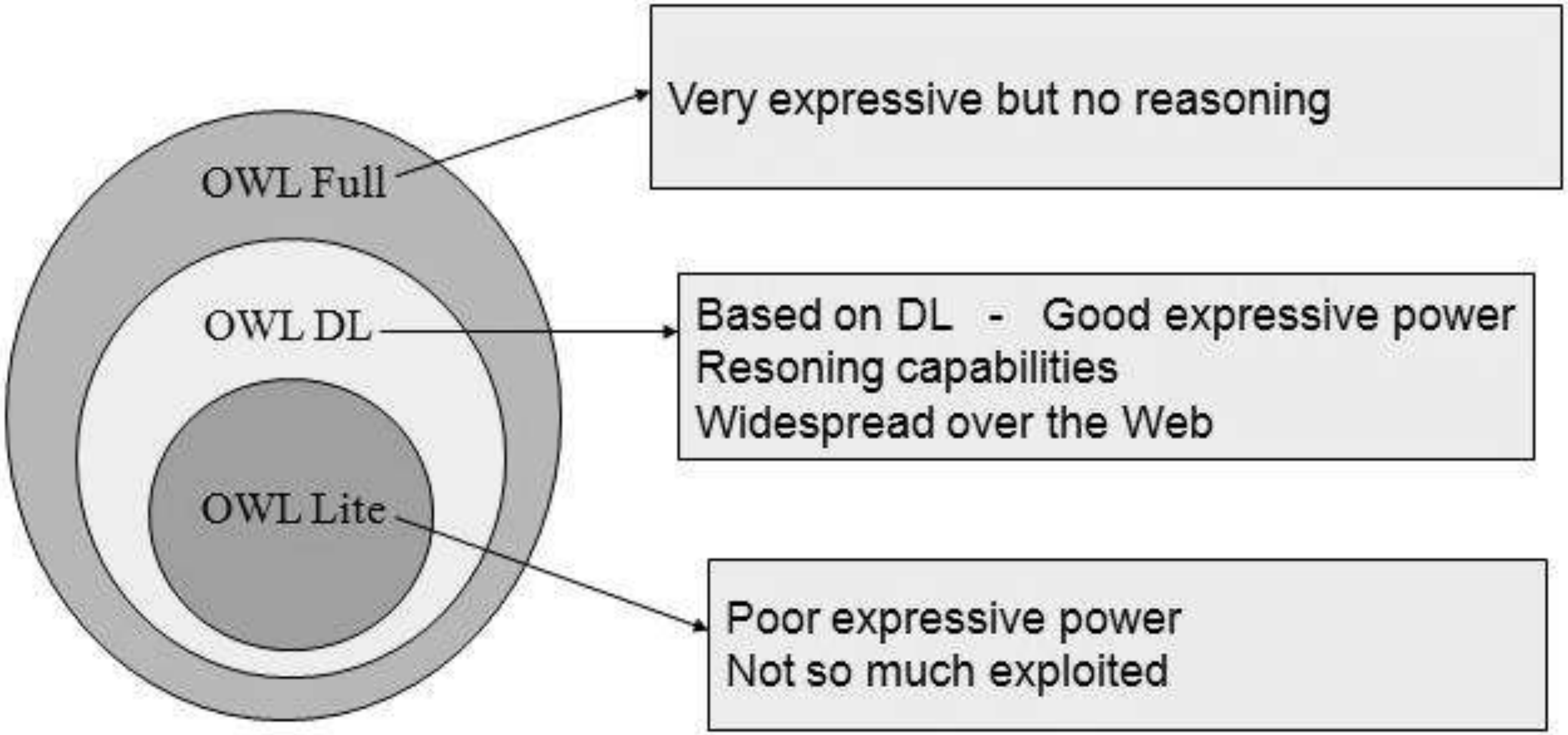
Other DL languages and their use

- **EL**
  - Fragment of OWL intended for Conceptual modelling (domain knowledge).
  - Example: SNOMED CT ontology ($3x10^5$ classes)
    - Large-scale healt
    - *Appendicitis ≡ Inf*

Person(?x) ^ EmployedBy(?x, ?company) ^ LocatedIn(?company, London) -> LondonBasedEmployee(?x)

- **RL**
  - Rule-based OWL, intended for inferencing
  - In practice SWRL (semantic web rule language) is used (not part of OWL)
- **QL**
  - Query-language, intended for ontology based data access (OBDA)
  - Queries roles and individuals (Abox)
  - SPARQL more general, queries also Tbox (concepts and their individuals)
    - Example: Retrieve all employees working in London offices who manage projects over $1M.

OWL flavours/versions



OWL flavours/versions

- OWL Full -> SROIQ(*D*)
  - Very high expressiveness, prohibitive reasoning
- OWL-DL -> SHOIN
  - Lower expressiveness, decidable
- OWL 2 -> SHOIQ
  - High expressiveness, very complex reasoning
- OWL-lite -> SHIF
  - Low expressiveness, high reasoning
- **Protégé supports SHOIN**

- $\mathcal{S}$ stands for $\mathcal{ALC}$ plus **role transitivity**,
- $\mathcal{H}$ stands for **role hierarchies**, i.e., **role inclusion** axioms,
- $\mathcal{O}$ stands for **nominals**, i.e., for closed classes with one element,
- $\mathcal{I}$ stands for **inverse roles**,
- $\mathcal{N}$ stands for **cardinality restrictions**,
- $\mathcal{D}$ stands for **datatypes**,
- $\mathcal{F}$ stands for **role functionality**,
- $\mathcal{Q}$ stands for **qualified cardinality restrictions**,
- $\mathcal{R}$ stands for **generalized role inclusion axioms**, and
- $\mathcal{E}$ stands for **existential role restrictions**.

Example of First Order Logic

Sentences

- Ander is Chinese
- Alona is Ander's daughter
- Panda likes Bamboo
- Children of football fans are football fans
- Football fans like summer

First order logic representation

- $\forall x, Chinese(x) \Longrightarrow Person(x)$
- $Chinese\ (Ander)$
- $hasDaughter(Ander, Alona)$
- $\forall x, y\ hasDaughter(x, y) \Longrightarrow childOf(y, x)$
- $\forall x, Panda(x) \Longrightarrow likes(x, Bamboo)$
- $\forall x, y, childOf(y, x)\ AND\ likes(x, Football) \Longrightarrow likes(y, Football)$
- $\forall x, likes(x, Football) \Longrightarrow likes(x, Summer)$

Example in OWL

```
@prefix : <http://hw.ac.uk/basques#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://hw.ac.uk/basques#> .

<http://hw.ac.uk/basques#> rdf:type owl:Ontology .
:Person rdf:type owl:Class.
:Sport rdf:type owl:Class.
:Seanon rdf:type owl:Class.
:Basque rdfs:subClassOf :Person.
:Woman rdfs:subClassOf :Person.
:Pelota rdfs:subClassOf :Sport.
```

Class definitions

```
:daughterOf rdf:type owl:ObjectProperty ;
        rdfs:domain :Woman .
:like rdf:type owl:ObjectProperty ;
        rdfs:domain :Person ;
        rdfs:range owl:Thing .
```

Object properties

```
:Ander rdf:type owl:NamedIndividual ,
        :Basque .
:Alona rdf:type owl:NamedIndividual ,
        :Person ;
        :daughterOf :Ander .
:summer rdf:type owl:NamedIndividual ,
        :Season ;
```

Individuals

```
Basque(?x) ^ Sport(?y) -> likeSport(?x, ?y)
childOf(?x, ?y) ^ likeSport(?y, ?s) -> likeSport(?x, ?s)
```

Use SWRL

The Second Rule (S2) in RDF Format

```
:x rdf:type <http://www.w3.org/2003/11/swrl#Variable> .

:y rdf:type <http://www.w3.org/2003/11/swrl#Variable> .

:s rdf:type <http://www.w3.org/2003/11/swrl#Variable> .
```

Variables

```
[ <http://swrl.stanford.edu/ontologies/3.3/swrla.owl#isRuleEnabled>
"true"^^xsd:boolean ;
rdfs:comment “Inherited hobby"^^xsd:string ;
rdfs:label "S2"^^xsd:string ;
rdf:type <http://www.w3.org/2003/11/swrl#Imp> ;
<http://www.w3.org/2003/11/swrl#body>
[ rdf:type <http://www.w3.org/2003/11/swrl#AtomList> ;
rdf:first [ rdf:type
<http://www.w3.org/2003/11/swrl#IndividualPropertyAtom> ;
<http://www.w3.org/2003/11/swrl#propertyPredicate> :childOf ;
<http://www.w3.org/2003/11/swrl#argument1> :x ;
<http://www.w3.org/2003/11/swrl#argument2> :y
] ;
```

Rule body

If ?x is a childOf ?y

```
rdf:rest [ rdf:type <http://www.w3.org/2003/11/swrl#AtomList> ;
  first [ rdf:type
  tp://www.w3.org/2003/11/swrl#IndividualPropertyAtom> ;
<http://www.w3.org/2003/11/swrl#propertyPredicate> :likeSport ;
<http://www.w3.org/2003/11/swrl#argument1> :y ;
<http://www.w3.org/2003/11/swrl#argument2> :s
] ; rdf:rest rdf:nil
]] ;
```

Rule body

```
<http://www.w3.org/2003/11/swrl#head>
[ rdf:type <http://www.w3.org/2003/11/swrl#AtomList> ;
rdf:first [ rdf:type
<http://www.w3.org/2003/11/swrl#IndividualPropertyAtom> ;
  //www.w3.org/2003/11/swrl#propertyPredicate> :likeSport ;
  /www.w3.org/2003/11/swrl#argument1> :x ;
  ttp://www.w3.org/2003/11/swrl#argument2> :s
] ; rdf:rest rdf:nil
] ] .
```

SWRL Example (human-readable)

Rule Name: S2 – Inherited Hobby

If:

- ?x is a child of ?y
- ?y likes sport ?s

Then:

- ?x also likes sport ?s

SWRL Syntax:

- childOf(?x, ?y) ^ likeSport(?y, ?s) → likeSport(?x, ?s)

OWL Advanced Features (Restrictions & Characteristics)

- Class restrictions
  - Class intersection (owl:and)
  - Class union (owl:or)
  - Class complement (owl:not)
  - Subclass of complex class
  - Enumerated class (list of individuals)
- Individual restrictions
  - Same individuals (owl:sameAs)
  - Different individuals (owl:differetFrom)
- Object Property restrictions
  - Existential restriction (owl:some)
  - Universal restriction (owl:only)
  - Equivalence (owl:some and owl:only)
  - Range value restriction (owl:value)
- Cardinality restrictions
  - Minimum range (owl:min)
  - Maximum range (owl:max)
  - Exact range (owl:exactly)

- Property characteristics
  - Inverse property (owl:inverseOf)
  - Symmetric (owl:SymmetricProperty, owl:AssymetricProperty)
  - Disjoint (owl:propertyDisjointWith)
  - Transitive (…:TransitiveProperty)
  - …inAxiom)
- …ns
  - …comparison

:hasGrandparent owl:propertyChainAxiom
(:hasParent :hasParent) .

Margherita Pizzas have toppings of Tomato and Mozzarella - moreover, they only have toppings of Tomato and Mozzarella

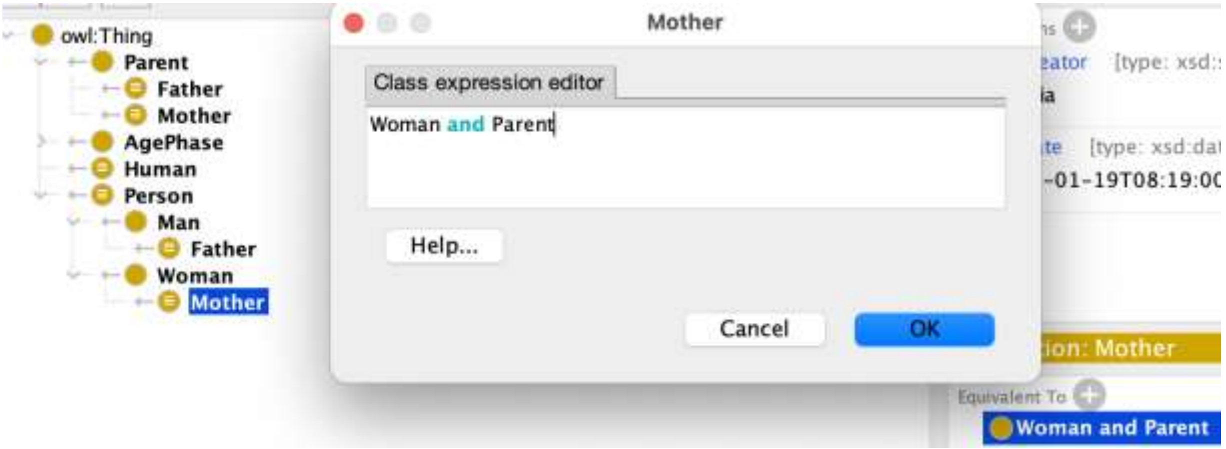
OWL: Intersection of classes

$$Woman(x)\ \boldsymbol{and}\ Parent(x) \Leftrightarrow \boldsymbol{Mother}(x)$$

- We can define a class as the intersection of two classes.
- Individuals of such class will be those belonging to both intersected classes
- Example Mother is Woman and Parent

```
:Mother rdf:type owl:Class ;
        owl:equivalentClass [ owl:intersectionOf ( :Parent
                                                   :Woman
                                                 ) ;
                              rdf:type owl:Class
                            ] ;
```

- Inference
  - Any individual mother will also be Parent



One-way vs Two-way statements (⊑ vs ≡)

- ➡️ One-way (implication / constraint)
  - A ⊑ B   means: If x is A → x must be B
  - ✅ From x:A infer x:B
  - ❌ From x:B cannot infer x:A
- 🔁 Two-way (equivalence / definition)
  - A ≡ B   means: A ⊑ B AND B ⊑ A (x is A ↔ x is B)
  - ✅ From x:A infer x:B
  - ✅ From x:B infer x:A   (enables auto-classification)
- 💻 Protégé (Manchester syntax)
  - A ⊑ B → Class: A SubClassOf: B
  - A ≡ B → Class: A EquivalentTo: B

OWL: Union of classes

$$Father(x)\ \textbf{or}\ Mother(x) \Leftrightarrow \boldsymbol{Parent(x)}$$

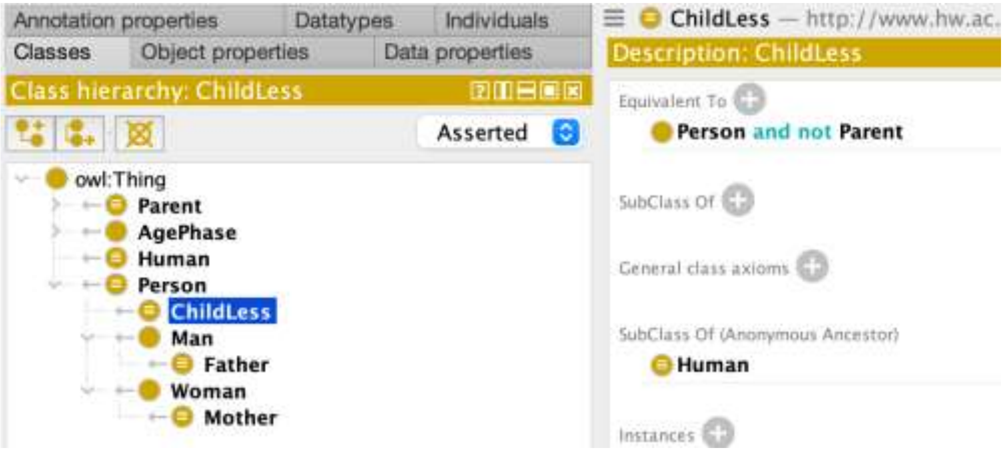
- A union class is the class of individuals that belong to at least one of union classes
- Example Parent is Mother or Father

```
:Parent rdf:type owl:Class ;
        owl:equivalentClass [ rdf:type owl:Class ;
                              owl:unionOf ( :Father
                                            :Mother
                                          )
                            ] ;
```

OWL: Complement classes (¬)

$$Person(x)\ and\ \mathbf{not}\ Parent(x) \Leftrightarrow \boldsymbol{ChildlessPerson(x)}$$

- A complement class is the class of individuals that belong to one class and not to another
- Example, person who have no children are defined as the complement of
  - Person and not Parent




```
:ChildlessPerson rdf:type owl:Class ;
                 owl:equivalentClass [ owl:intersectionOf ( :Person
                                                            [ rdf:type owl:Class ;
                                                              owl:complementOf :Parent
                                                            ]
                                                          ) ;
                                       rdf:type owl:Class
                                     ] ;
```

OWL: Subclass of complex class

$$Parent(x)\ \ \boldsymbol{and}\ \ Man(x) \Longrightarrow \boldsymbol{GrandFather(x)}$$

- A class can be
- Individuals of
  also individua
- Example
  - A GrandF

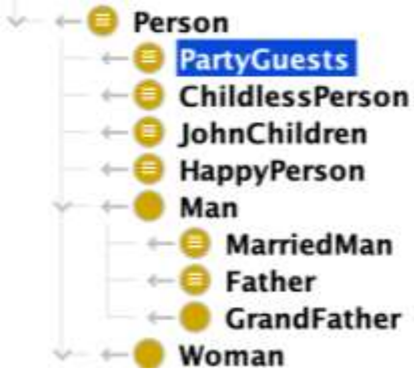
$$GrandFather(\boldsymbol{x}) \Longrightarrow\ Parent(x)\ \ \boldsymbol{and}\ \ Man(x)$$

$$\mathrm{GrandFather(x)} \equiv \mathrm{Man(x)} \land \exists \mathrm{y}(\mathrm{hasChild(x,y)} \land \mathrm{Parent(y)})$$

- This expr
  sufficien
  - ✓ If
  - ✗ If x is a Man AND a Parent → x is NOT necessarily a GrandFather

OWL: Enumerated class (owl:oneOf)

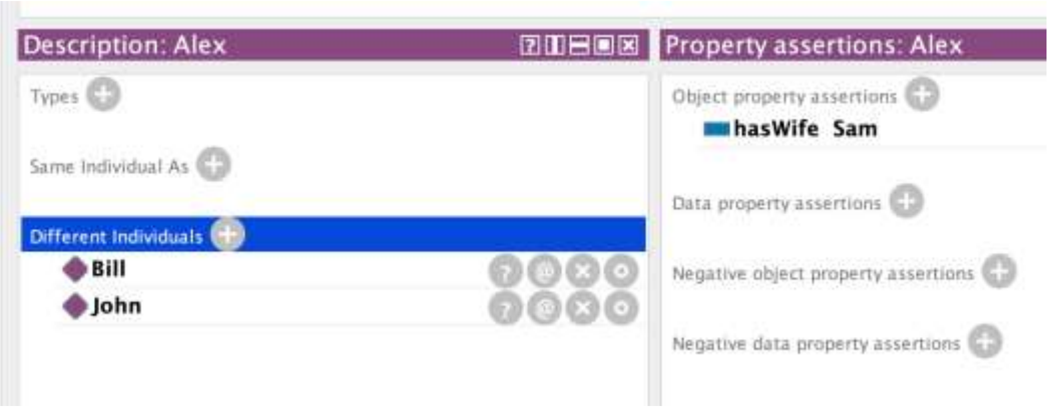
- Define a class by listing its exact members (a “closed list”).
- Example:
  - Bill, John and Mary are members of the class PartyGuests




```
:PartyGuests rdf:type owl:Class ;
             owl:equivalentClass [ rdf:type owl:Class ;
                                   owl:oneOf ( :Bill
                                               :John
                                               :Mary
                                             )
                                 ] ;
```

OWL: Distinct individuals (owl:differentFrom)

- OWL **does NOT** assume different names = different entities (no “Unique Name Assumption”).
- Example, Alex and John might be considered the same
- To avoid this, we assert that they are different
  - Alex different from John and from Bill




```
[ rdf:type owl:AllDifferent ;
  owl:distinctMembers ( :Alex
                        :Bill
                      )
] .
```

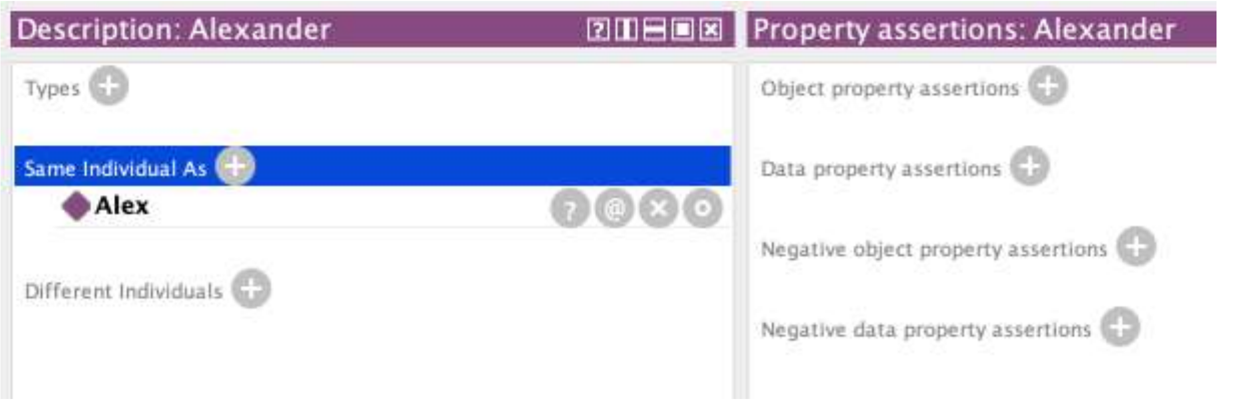
```
[ rdf:type owl:AllDifferent ;
  owl:distinctMembers ( :Alex
                        :John
                      )
] .
```

OWL: Same individuals (owl:sameAs)

- With OWL we can assert that two individuals are the same
- Example: if Alex and Alexander are the same

```
:Alex rdf:type owl:NamedIndividual ;
       owl:sameAs :Alexander ;
       :hasWife :Sam ;
```

- Owl:sameAs combines all properties of two instances
  - The two become indistinguishable
- This might not be the behaviour sought for your application
  - In such a case use skos:exactMatch or skos:closeMatch

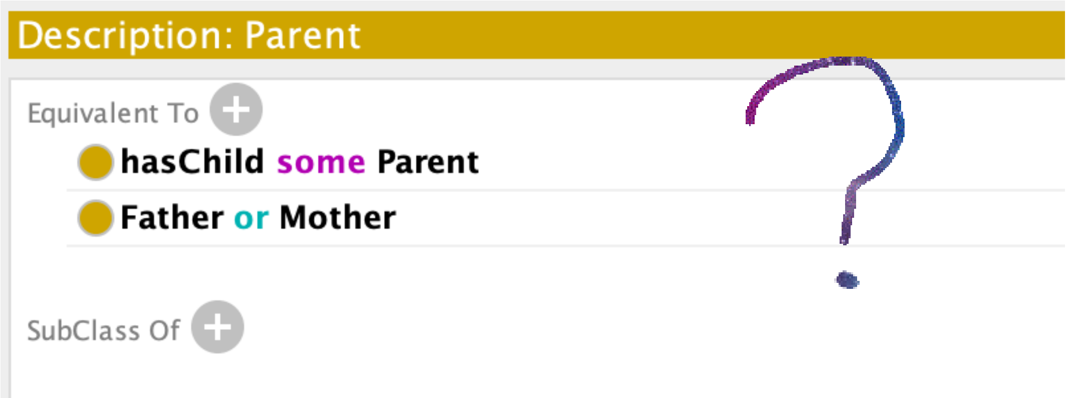


Property restrictions- existential quantification (some / ∃)

$$C(x) \Longrightarrow \exists y, p(x, y)$$

- P some C = “there exists at least 1 value of property P that is in class C”

```
:Parent rdf:type owl:Class ;
    owl:equivalentClass [
        rdf:type owl:Class ;
        owl:intersectionOf (
            [ rdf:type owl:Class ;
              owl:unionOf ( :Father :Mother )
            ]
            [ rdf:type owl:Restriction ;
              owl:onProperty :hasChild ;
              owl:someValuesFrom :Person
            ]
        )
    ] .
```



- This is expressed as: *every parent has at least one child who is a person*

Property restrictions- Universal quantification

'only' does NOT guarantee existence of children.
A HappyPerson with no stated children still satisfies: hasChild only HappyPerson.

$\forall y, p(x,y)$

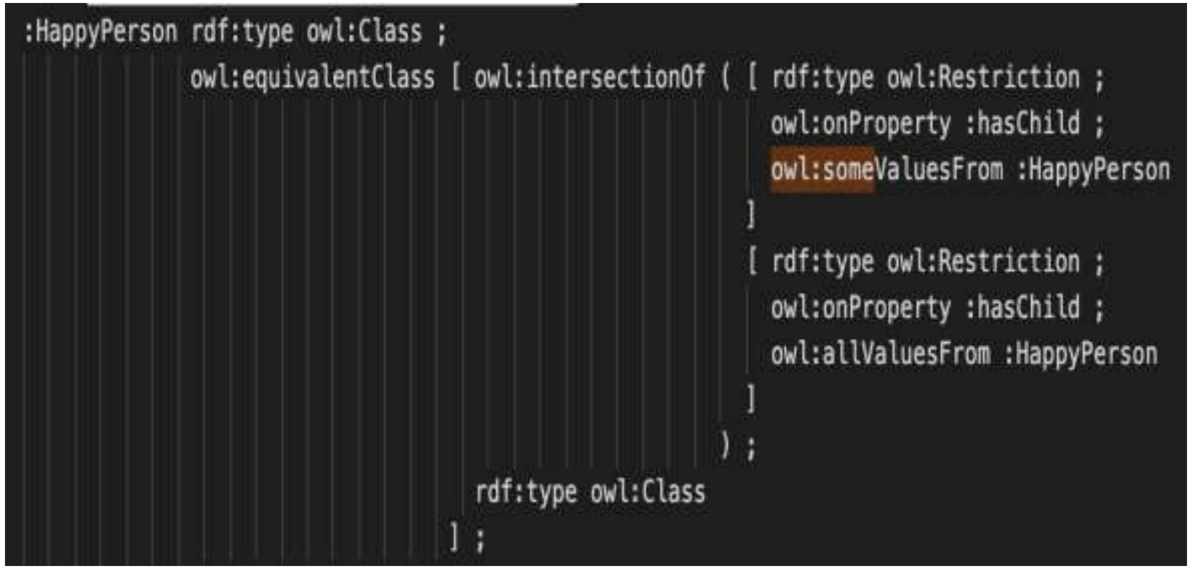
- A person is happy only if their children are happy
- HappyPerson ⊑ hasChild only HappyPerson
- HappyPerson(x) → ∀y (hasChild(x,y) → HappyPerson(y))

```
:HappyPerson rdf:type owl:Class ;
    owl:equivalentClass [
        owl:intersectionOf (
            [ rdf:type owl:Restriction ;
              owl:onProperty :hasChild ;
              owl:allValuesFrom :HappyPerson
            ]
        )
    ] .
```

- This is expressed as: children of *happy person must all be happy.*

Property restrictions – Closure (some + only)

- P some X = at least one P-value is in X
- P only X = all P-values (if any) are in X
- Together = ***at least*** one exists AND all are of class X
- Example:
  - HappyPerson ⊑ (hasChild some HappyPerson) ⊓ (hasChild only HappyPerson)
  - = A happy person has at least one child, and all their children are happy.

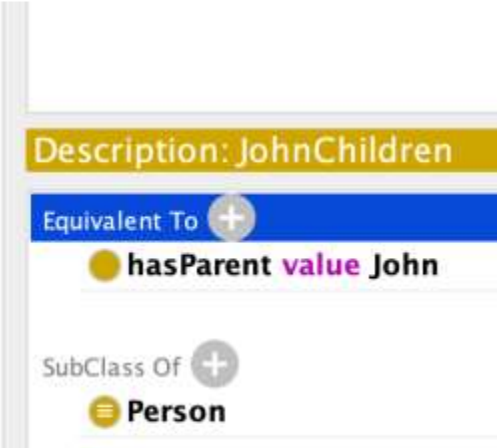


Property restrictions – Closure (some + only) cont.

- With using **only:**
  - **MeatLoversPizza** ⊑ (hasTopping only Meat)
    - BUT a Plain Crust (no toppings) is technically a Meat Lovers pizza because it has "only meat”.
- Correct version?
  - **MeatLoversPizza** ⊑ (hasTopping some Meat) **⊓** (hasTopping only Meat)
    - Some: *You must have at least one meat topping*.
    - Only: *You are forbidden from having anything other than meat*.

Property restrictions- Restriction on individual (value/hasValue)

- P value a = To be in this group, you must be connected to ‘a’.
- Example:
  - JohnChildren = people whose parent is John
    - *hasParent value John*
      - “Ali
  - class: Ita
    - *madel*
      - “Gu



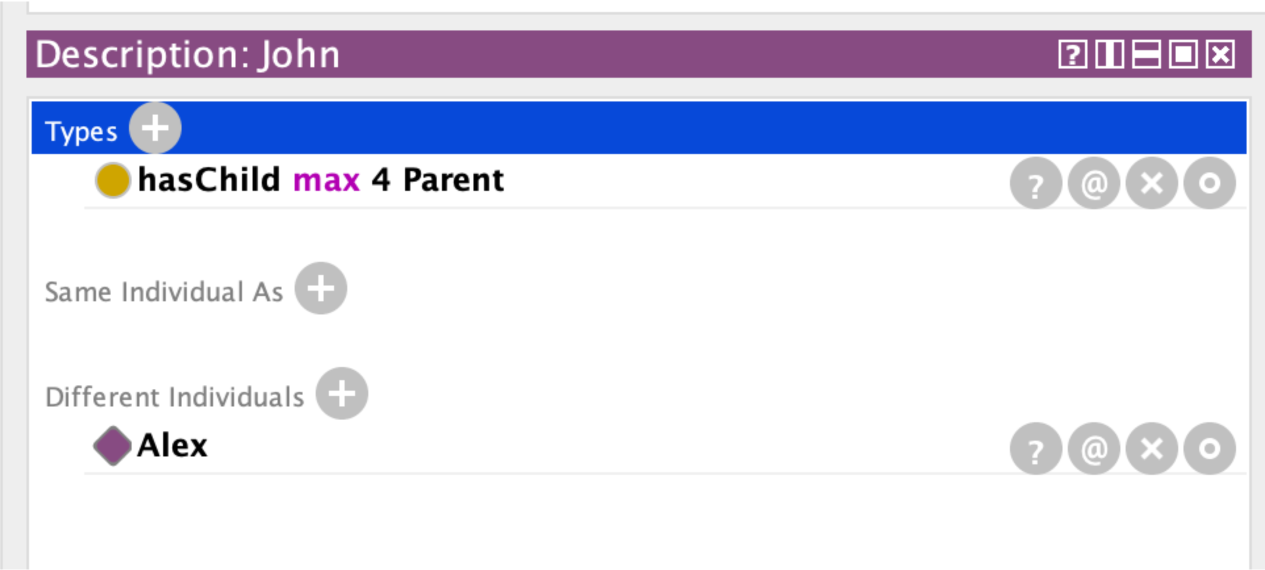
What the reasoner can infer (if using EquivalentTo)?

✓ If Mary hasParent John → Mary is a JohnChildren

✓ If Mary is a JohnChildren → Mary hasParent John

OWL: Cardinality restriction (max)

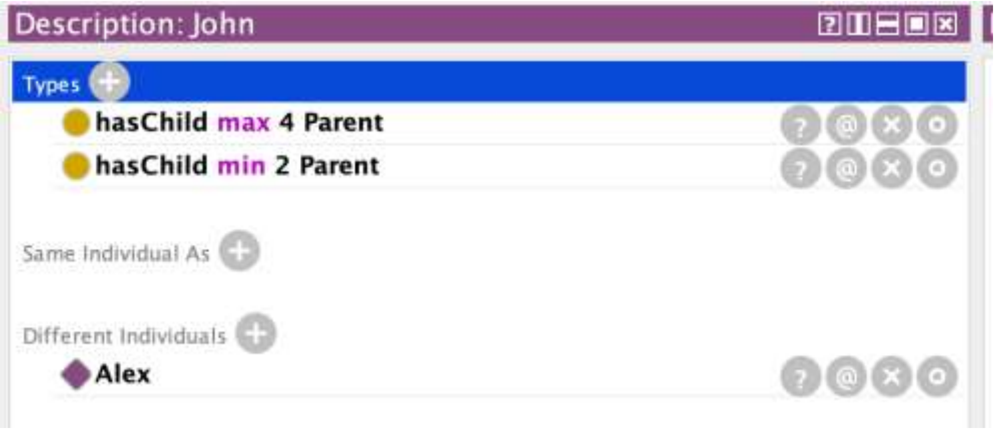
- For upper bound/limit.
- P max *n* C = **at most** ***n*** DIFFERENT/DISTINCT values of property P are in class C
- Example:
  - John has at most 4 children who are parents
- Qualified vs unqualified:
  - *hasChild max 4*
    - at most 4 children (any type)
  - *hasChild max 4 Parent*
    - at most 4 children that are Parents




```
:John rdf:type owl:NamedIndividual ,
[ rdf:type owl:Restriction ;
        owl:onProperty :hasChild ;
        owl:maxQualifiedCardinality "4"^^xsd:nonNegativeInteger ;
        owl:onClass :Parent
      ] ,
```

OWL: Cardinality restriction (min)

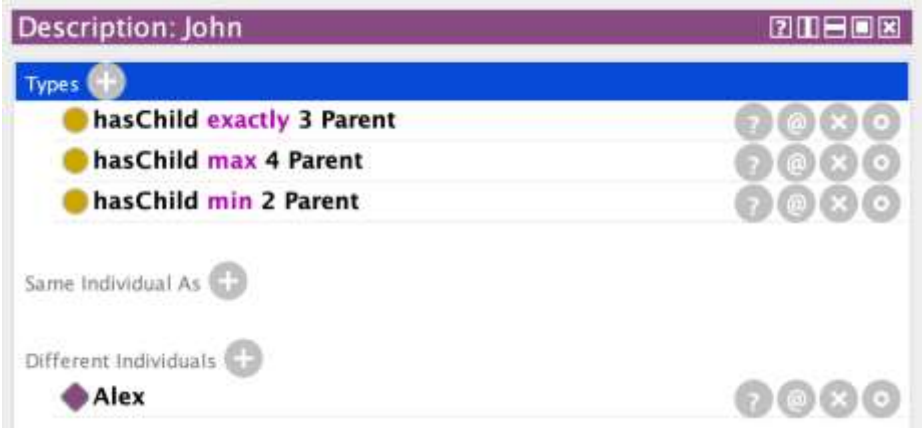
- P min n C = **at least** *n* values of property P are in class C (= at least n DIFFERENT individuals in C)
- Example
  - John has at least 2 children who are parents
    - → hasChild min 2 Parent
- Qualified vs unqualified?




```
:John rdf:type owl:NamedIndividual ,
               [ rdf:type owl:Restriction ;
                 owl:onProperty :hasChild ;
                 owl:minQualifiedCardinality "2"^^xsd:nonNegativeInteger ;
                 owl:onClass :Parent
               ] ,
```

OWL: Cardinality restriction (exactly)

- We can use ***owl:qualifiedCardinality*** to specify the **exact number** of individuals related by a certain property that also ***belong to a specific class***.
- P exactly n C = (P min n C) AND (P max n C)
  - exactly *n* **DIFFERENT** values of property P that are in class C
- Example
  - John has exactly 3 children who are of type Parent.
- Qualified vs unqualified.



Using exactly, min and max in an ontology increases its complexity

```
:John rdf:type owl:NamedIndividual ,
              [ rdf:type owl:Restriction ;
                owl:onProperty :hasChild ;
                owl:qualifiedCardinality "3"^^xsd:nonNegativeInteger ;
                owl:onClass :Parent
              ] ,
```

OWL reasoning notes…

- Data: John has 3 children…
  - Child A
  - Child B
  - Child 3
- Question: Does John have 4 children?
- OWL/Semantic is under OWA.

OWL: Cardinality restriction (unqualified)

- Unqualified cardinality = counting WITHOUT a type constraint
- P exactly n = exactly n distinct values for property P (any type)
  - (similar idea for P min n, P max n)
- Example
  - Define for a “fully married man” that the number of wives is 4


```
:MarriedMan rdf:type owl:Class ;
            owl:equivalentClass [ rdf:type owl:Restriction ;
                                  owl:onProperty :hasWife ;
                                  owl:cardinality "4"^^xsd:nonNegativeInteger
                                ] ;
            rdfs:subClassOf :Man .
```

OWL: Cardinality restriction (qualified vs unqualified)

Cardinality can count:

1. fillers of a **specific** TYPE (qualified), or
2. fillers of **ANY** type (unqualified)

Qualified cardinality (type matters):

- Person $\sqsubseteq$ hasPet min 1 Cat

Unqualified cardinality (type does NOT matter):

- Person $\sqsubseteq$ hasPet min 1

OWL: property characteristics - Inverse (owl:inverseOf)

- Edge reversal in a graph:
  - If ***p*** is the inverse of ***q***, then:
    - ***p***(x, y) $\leftrightarrow$ ***q***(y, x)
- Example
  - *hasHusband is the inverse property of hasWife*
  - *hasParent inverseOf hasChild*
    - *hasParent(Alice, Bob)* $\rightarrow$ *hasChild(Bob, Alice)*
    - *hasChild(Bob, Alice)* $\rightarrow$ *hasParent(Alice, Bob)*

```
:hasHusband rdf:type owl:ObjectProperty ;
            rdfs:subPropertyOf :hasSpouse ;
            owl:inverseOf :hasWife ;
```

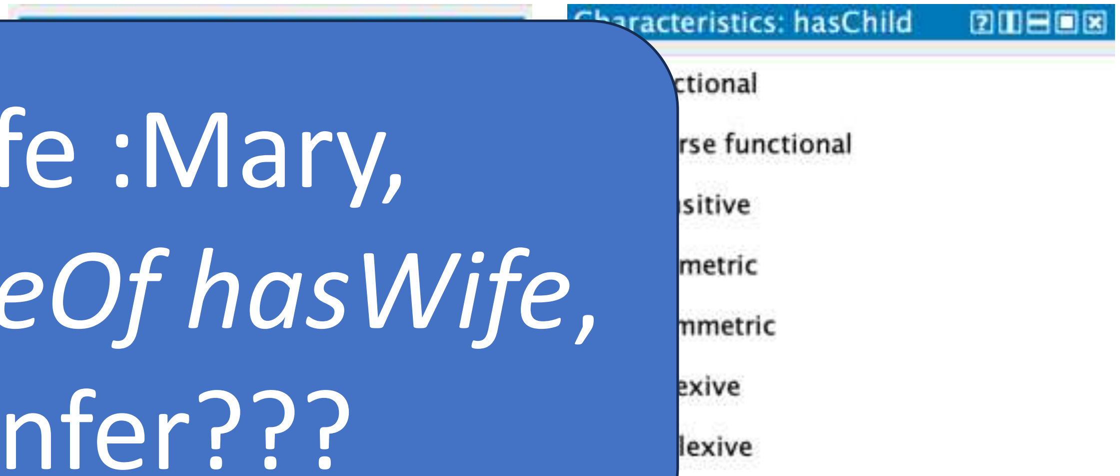
OWL: property characteristics (symmetry/asymmetric)

Symmetric property (two-way):

- If P(x, y)
  - the re
    direct
- Example
  - hasSp
    - ha
      hasSpouse(Mary, John)

```
:hasSpouse rdf:type owl:ObjectProperty ,
                    owl:SymmetricProperty ;
```

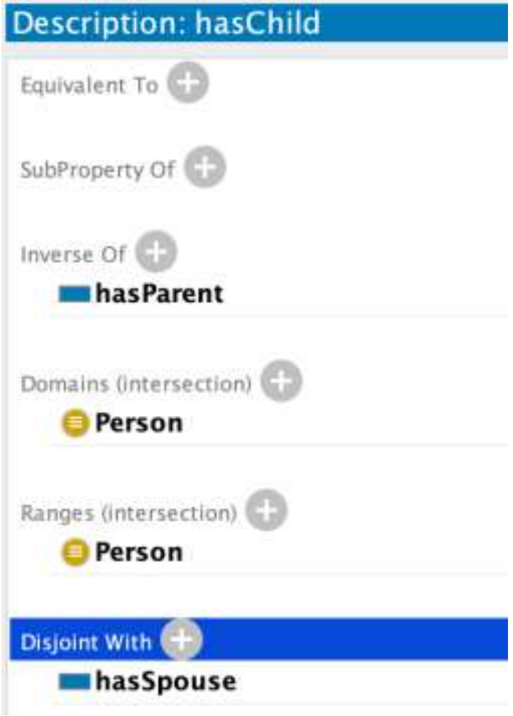
- hasChild is asymmetric




```
:hasChild rdf:type owl:ObjectProperty ;
          owl:inverseOf :hasParent ;
          rdf:type owl:AsymmetricProperty ;
          rdfs:domain :Person ;
          rdfs:range :Person ;
```

If :John :hasWife :Mary,
hasHusband inverseOf hasWife,
what can we infer???

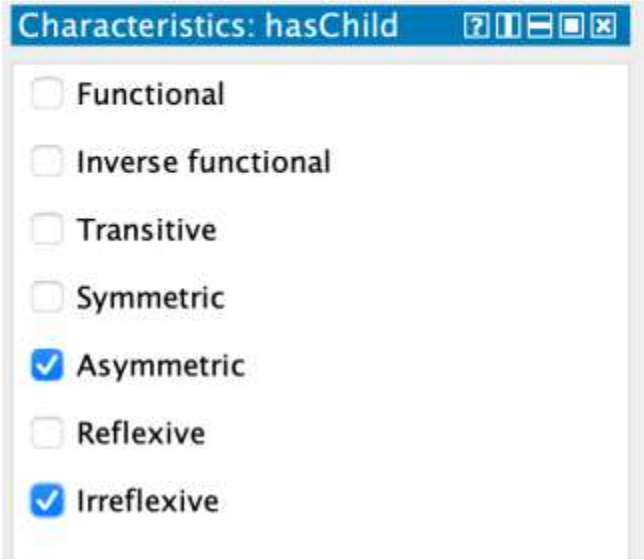
OWL: property characteristics (disjoint)



- Two properties P and Q are disjoint if they can NEVER hold for the same pair (x, y).
  - Formal: NOT (P(x, y) AND Q(x, y))
- Example
  - *hasChild disjointWith hasSpouse*
    - hasChild and hasSpouse are disjoint
- So you cannot have:
  - hasChild(John, Mary) AND hasSpouse(John, Mary)
- What if?
  - hasChild(John, Mary) AND hasSpouse(John, Anna)

```
:hasChild rdf:type owl:ObjectProperty ;
          owl:inverseOf :hasParent ;
          rdf:type owl:AsymmetricProperty ;
          rdfs:domain :Person ;
          rdfs:range :Person ;
          owl:propertyDisjointWith :hasSpouse ;
```

OWL: property characteristics (reflexive/irreflexive)



- A property is reflexive if the domain and range can be the same.
  - The property can be applied to yourself
  - For every individual x: P(x, x)
- Irreflexive is when the range and domain **cannot** be the same.
- Example: assuming there is only one John (same IRI)
  - *knows(John, John)* ✓
  - *isSameNationalityAs(John, John)* ✓
  - *hasChild(John, John)* is irreflexive !
  - *isTallerThan(John, John)?*

```
:hasChild rdf:type owl:ObjectProperty ;
          owl:inverseOf :hasParent ;
          rdf:type owl:AsymmetricProperty ,
                   owl:IrreflexiveProperty ;
```

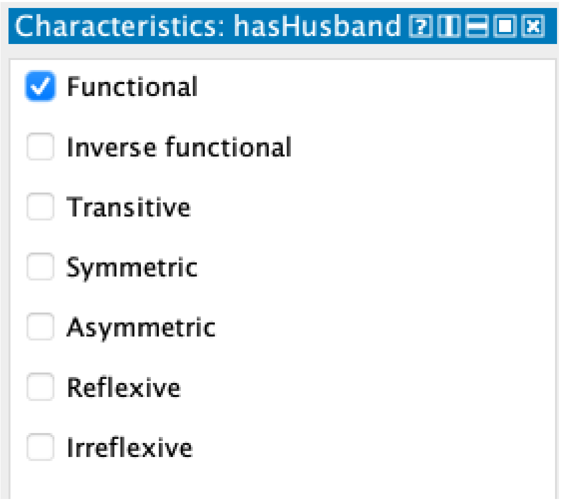
OWL: property characteristics (functional vs. inverse functional)

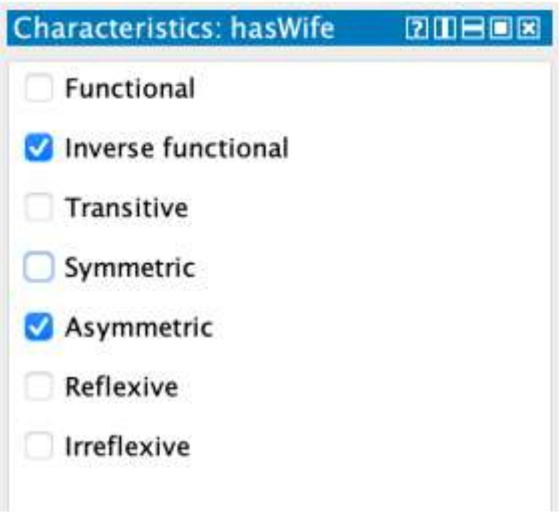
- Functional means one individual form the domain can only be related to one individual from the range.
- Inverse functional is the opposite
- Example
  - hasHusband is functional

```
:hasHusband rdf:type owl:ObjectProperty ;
            rdfs:subPropertyOf :hasSpouse ;
            owl:inverseOf :hasWife ;
            rdf:type owl:FunctionalProperty ;
```

  - hasWife is inverseFunctional is irreflexive

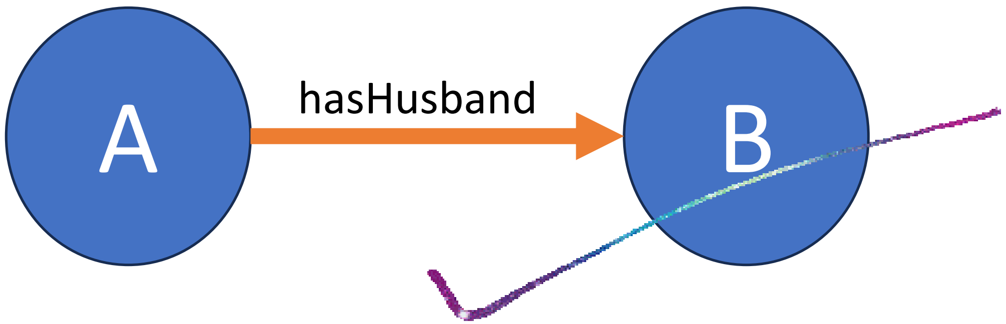
```
:hasWife rdf:type owl:ObjectProperty ;
         rdfs:subPropertyOf :hasSpouse ;
         rdf:type owl:InverseFunctionalProperty ,
                  owl:AsymmetricProperty ;
```

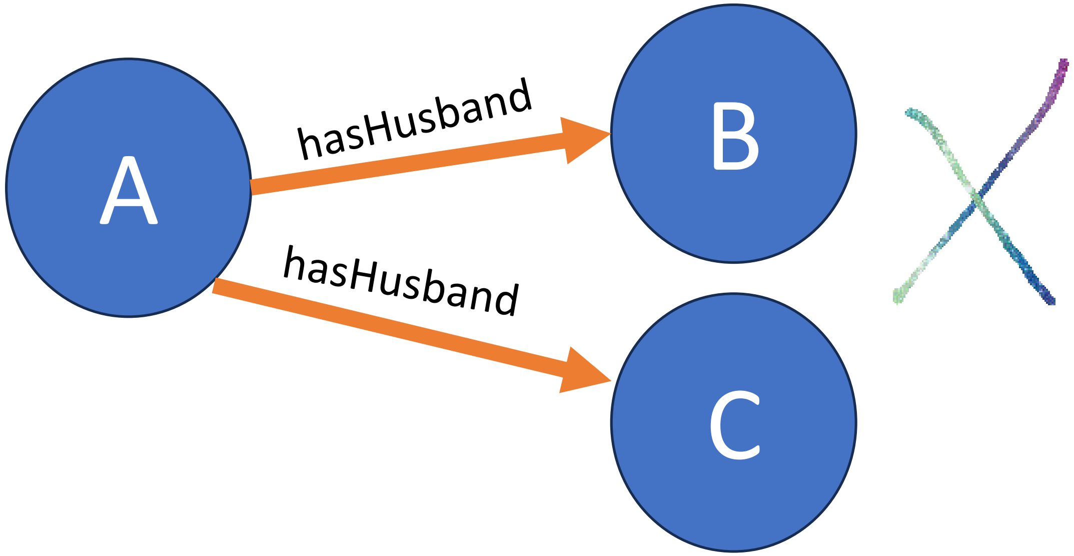




OWL: property characteristics (functional vs. inverse functional) cont.

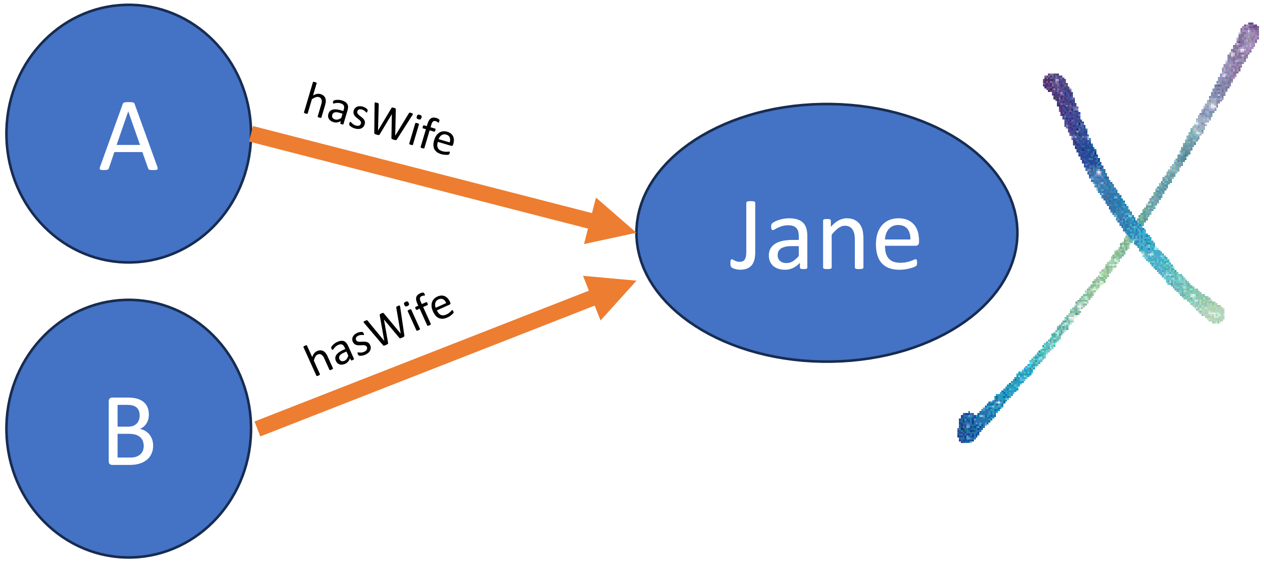
- Functional = One person → One value.
  - "Source" (the tail of the arrow) can only have **one** arrow coming out of it
- Inverse functional = One value ← One person
  - "Target" (the head of the arrow) can only have **one** arrow pointing into it
- *hasHusband is **functional***





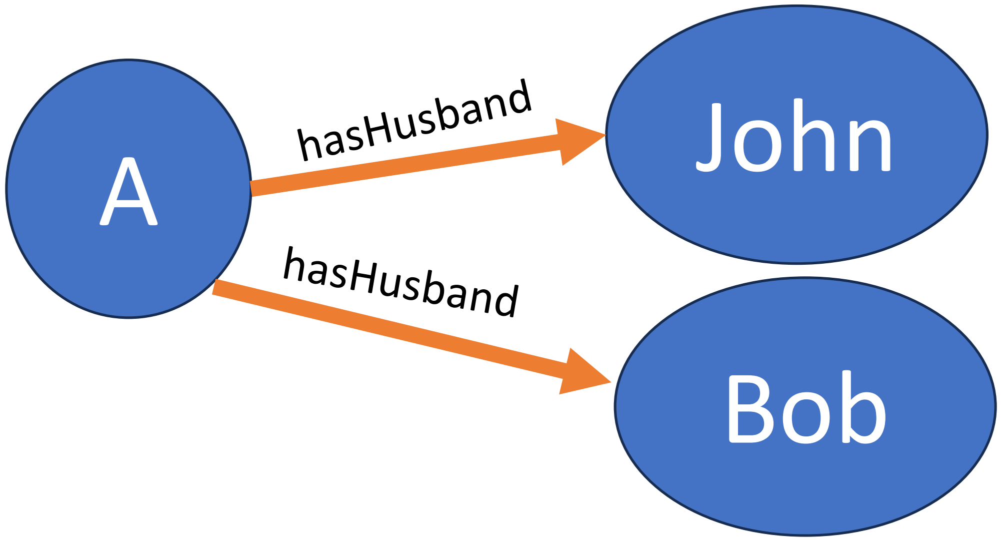
OWL: property characteristics (functional vs. inverse functional) cont.

- "Target" (the head of the arrow) can only have one arrow pointing into it
- If *hasWife is* ***inverse functional…***



OWL: property characteristics (functional vs. inverse functional) cont.

- *hasHusband is functional*



OWL: property characteristics (functional vs. inverse functional) cont.

If *P* is functional, then *P* is inverse functional too?

- Property: *hasBirthCity*
  - *Functional*? Yes! A person has only one birth city (Source → 1 Target).
    - "John hasBirthCity KL"
  - *Inverse functional?* No! Millions of people can be born in the same city.

If *P* is inverse functional, then *P* is functional too?

- Property: *ownsCarWithPlate*
  - *Inverse Functional*? Yes! A specific license plate (the target) belongs to only one owner (the source). (1 Source ← Target).
    - "Zihau ownsCarWithPlate → ABC123"
  - *Functional?* No! One person can own more than one cars with different plates.

OWL: property characteristics (functional vs. inverse functional) cont.

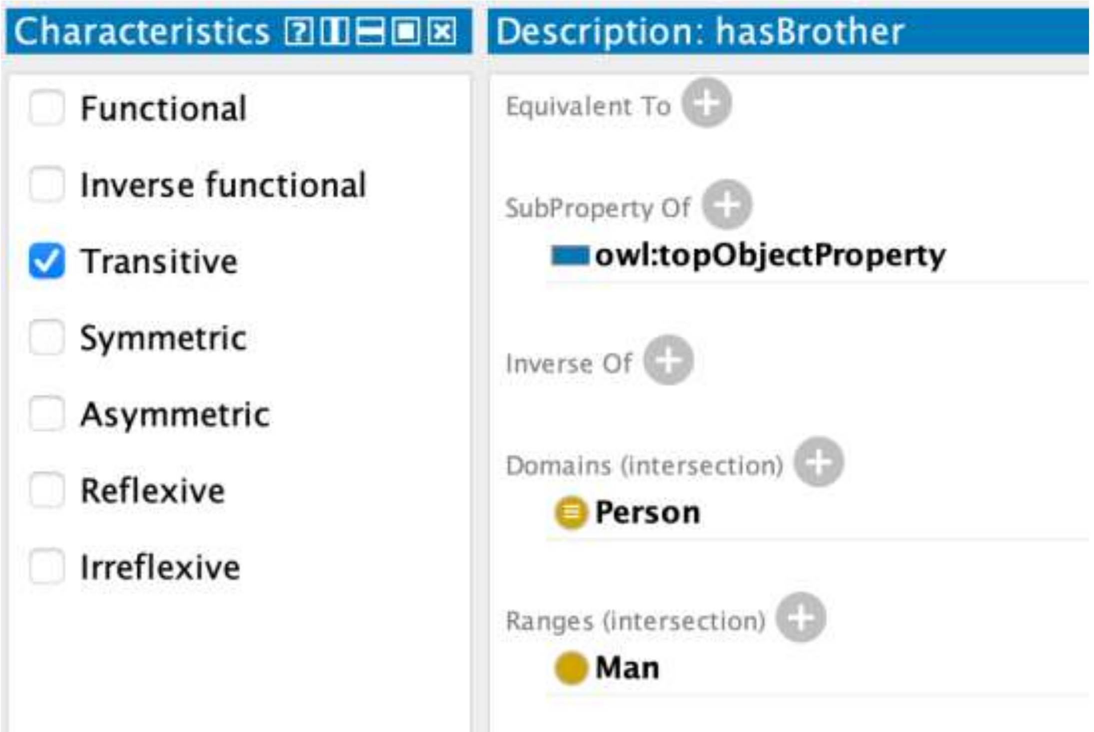
Then can we set both functional and inverse functional together?

- Yes! The 1:1 relationship.

- Property: *hasHusband* (in a strictly monogamous model).
- ***Functional***: A wife has only one husband.
- ***Inverse Functional***: A husband has only one wife.
- The Result? A perfect "marriage" of the two rules where 1 source matches exactly 1 target.

OWL: property characteristics (transitive)

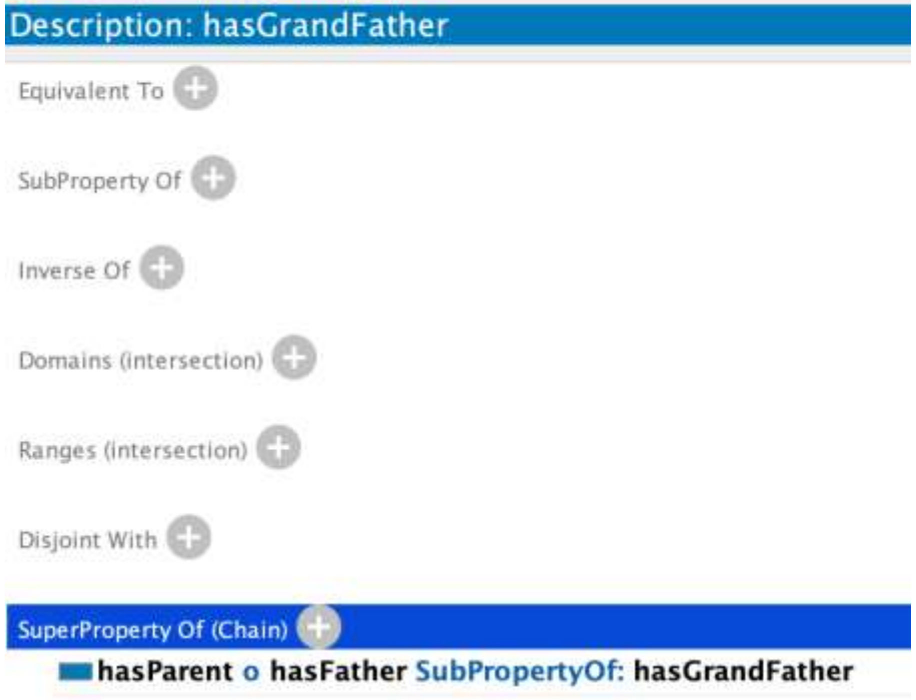
- if ***x*** is related to ***y***, and ***y*** is related to ***z***, then ***x*** is related to ***z.***
- Example:
  - isOlder:
    - *John isOlder Charlie*
    - *Charlie isOlder Bob*
    - Then… *John isOlder Bob*




```
:hasBrother rdf:type owl:ObjectProperty ;
            rdfs:subPropertyOf owl:topObjectProperty ;
            rdf:type owl:TransitiveProperty ;
            rdfs:domain :Person ;
            rdfs:range :Man ;
```

OWL: property characteristics (property chain)

- We can define a property as a chain (sequence) of other properties
  - If p(x, y) AND q(y, z) then r(x, z)
- Example, hasGrandFather can be defined as the chain of hasParent and hasFather
  - hasParent(x, y) AND hasFather(y, z) ⇒ hasGrandFather(x, z)
- Note: Property chain is not "transitive"! It creates a NEW property!




```
:hasGrandFather rdf:type owl:ObjectProperty ;
                owl:propertyChainAxiom ( :hasParent
                                         :hasFather
                                       ) ;
```

Data properties and restrictions

Data values may be untyped or typed (eg int, boolean, float etc). The types available will depend on tool support, but will include those specified in the XSD recommendation.

Constants can be expressed without type by just enclosing them in double quotes, or with type

```
hasAge value "21"^^long
```

Usage of these datatypes in more general expressions is possible through their shortened name

```
hasAge some int
```

Several additional XSD facets can also be used to create new datatypes

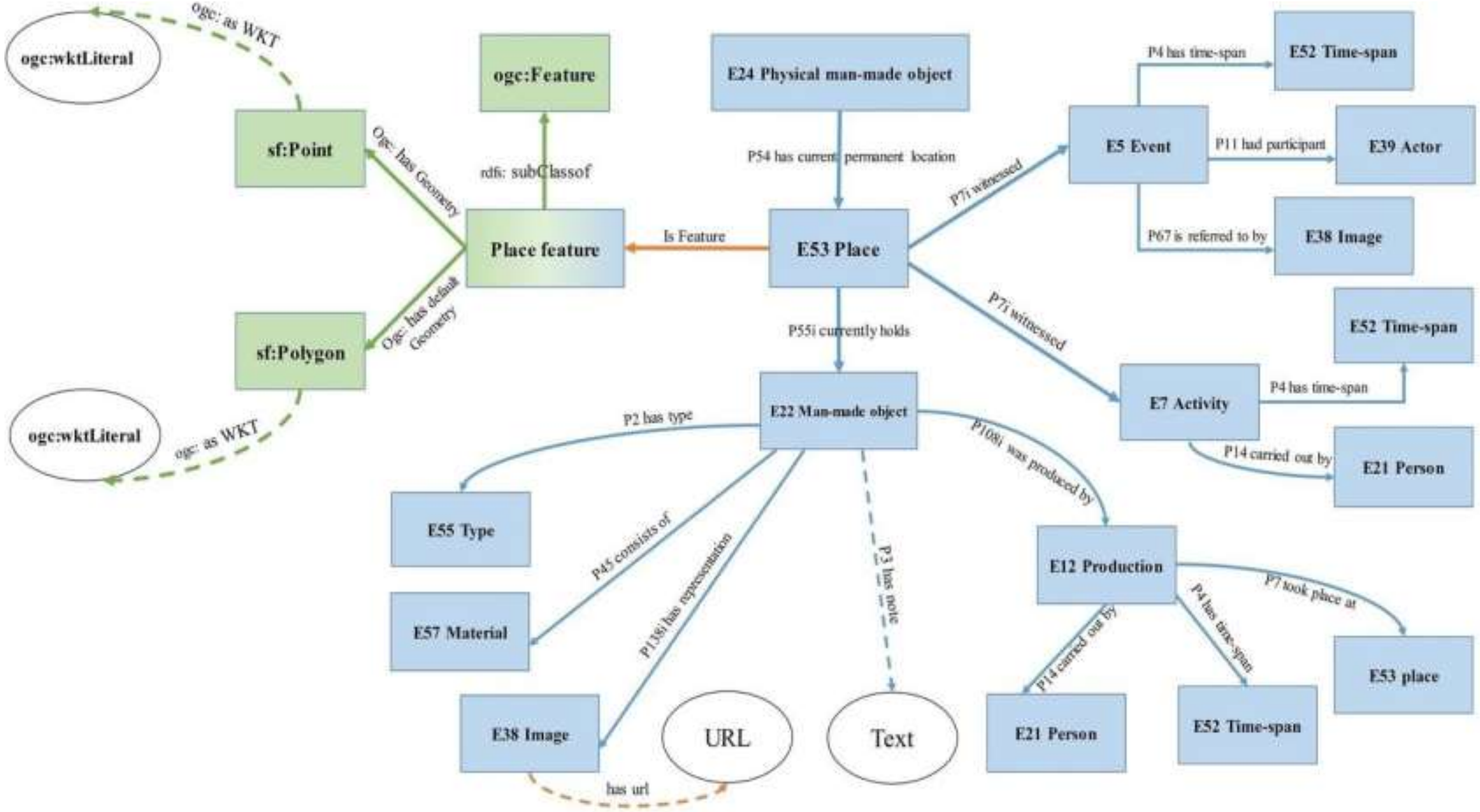
```
Person and hasAge some int[>= 65]
```

Multiple facets can also be used. For example, when wishing to express numeric ranges

```
Person and hasAge some int[>= 18, <= 30]
```

XSD facet	Meaning
< x, <= x	less than, less than or equal to x (more info)
> x, >= x	greater than, greater than or equal to x (more info)
length x	For strings, the number of characters must be equal to x (more info)
maxLength x	For strings, the number of characters must be less than or equal to x (more info)
minLength x	For strings, the number of characters must be greater than or equal to x (more info)
pattern regexp	The lexical representation of the value must match the regular expression, regexp (more info)
totalDigits x	Number can be expressed in x characters (more info)
fractionDigits x	Part of the number to the right of the decimal place can be expressed in x characters (more info)

Please note: some xsd datatypes and facets may not be supported by particular reasoners at this current time.
A report on the current status of reasoner implementations is available



- How can Knowledge Graphs (KGs) enhanced Large Language Models (LLMs)?
- How can LLMs support and enhance Knowledge Graphs?

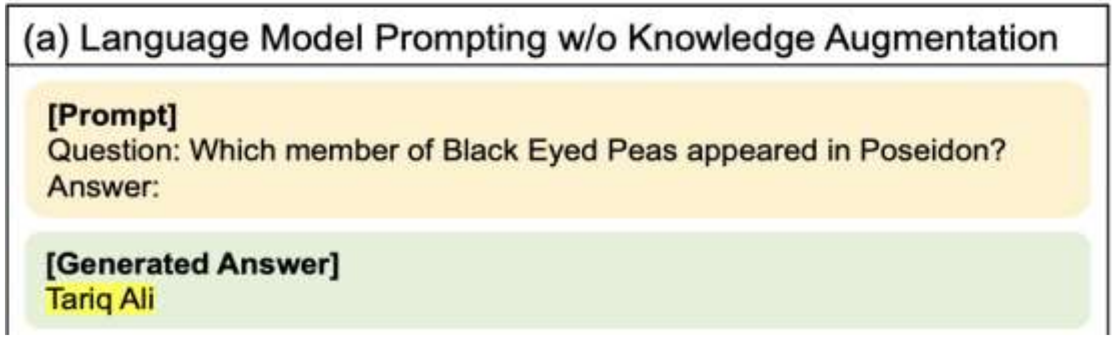
LLMs Empowered by KGs

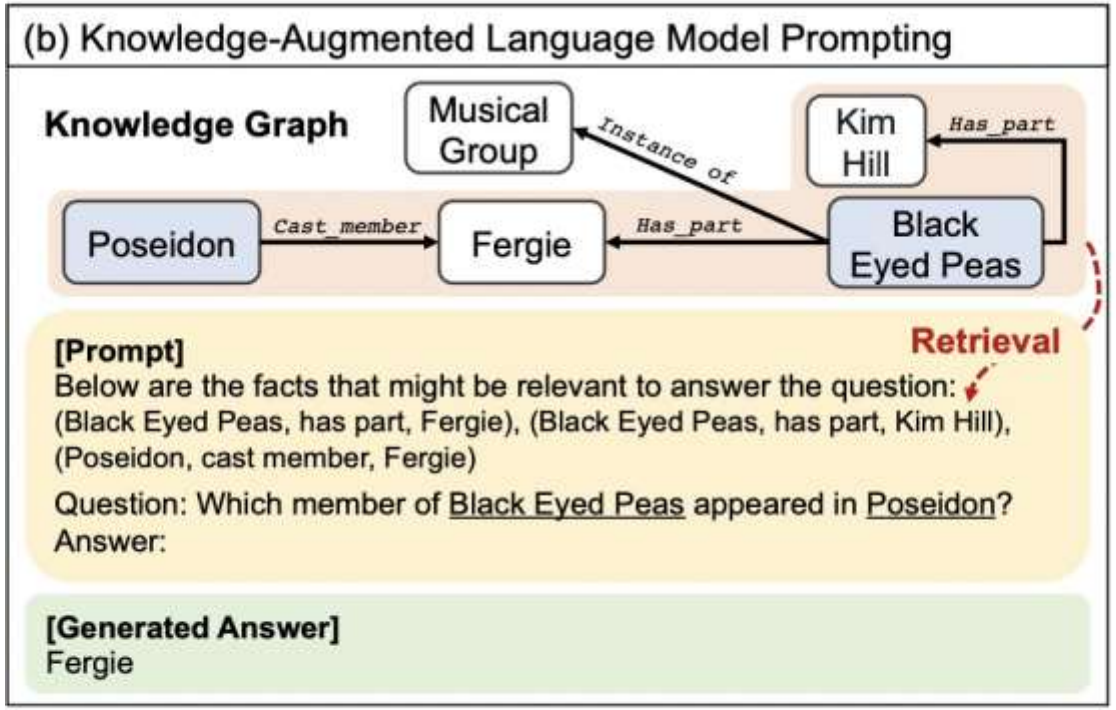
Knowledge injection:

- Limitations of LLMs:
  - generate inaccurate or nonsensical information (hallucinations),
  - Lack of domain specific knowledge

Knowledge-Augmented Language Model PromptING (KAPING)

- Retrieved relevant facts from a KG.
- Add these facts to the user's input question.
- Construct a richer prompt for the LLM.

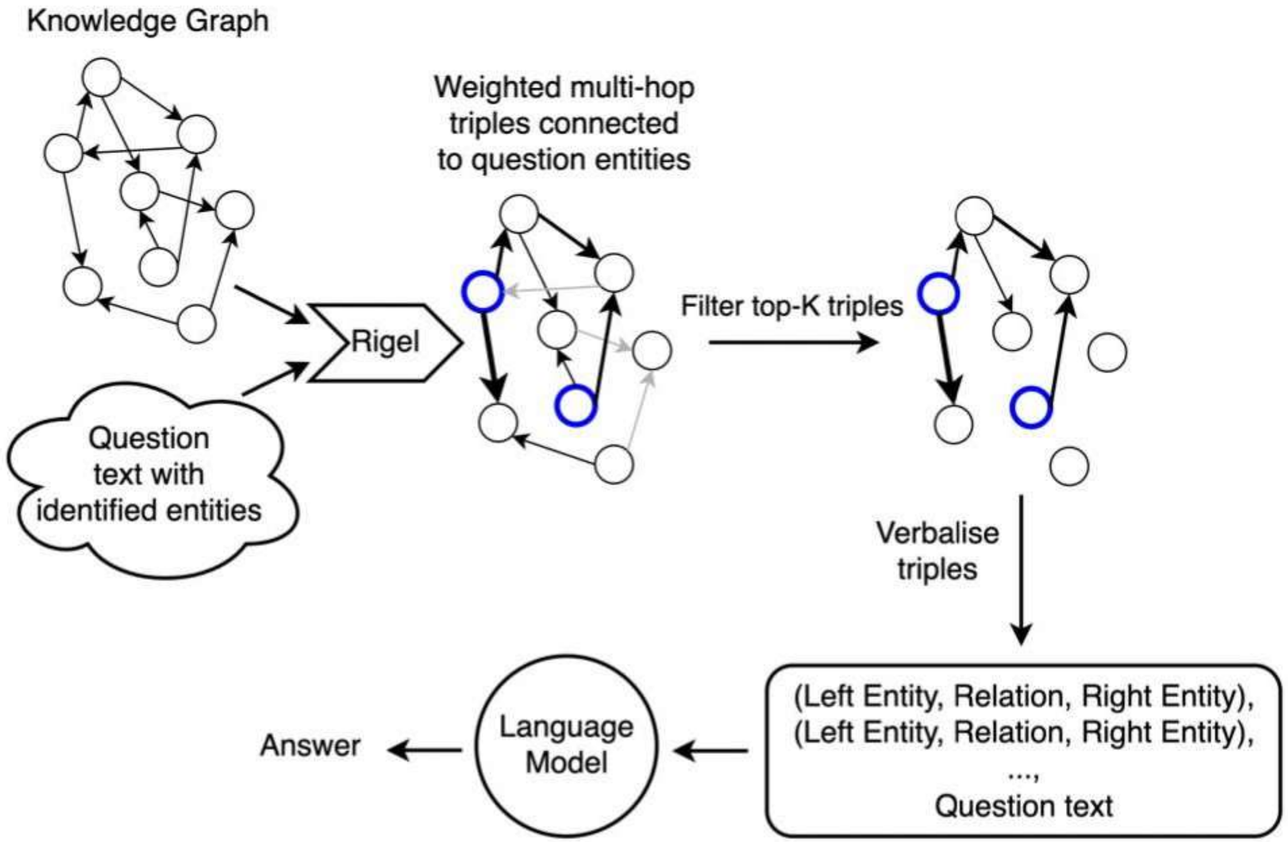




LLMs Empowered by KGs

- LLMs often struggle with complex, multi-step reasoning because they rely on probabilistic text generation.

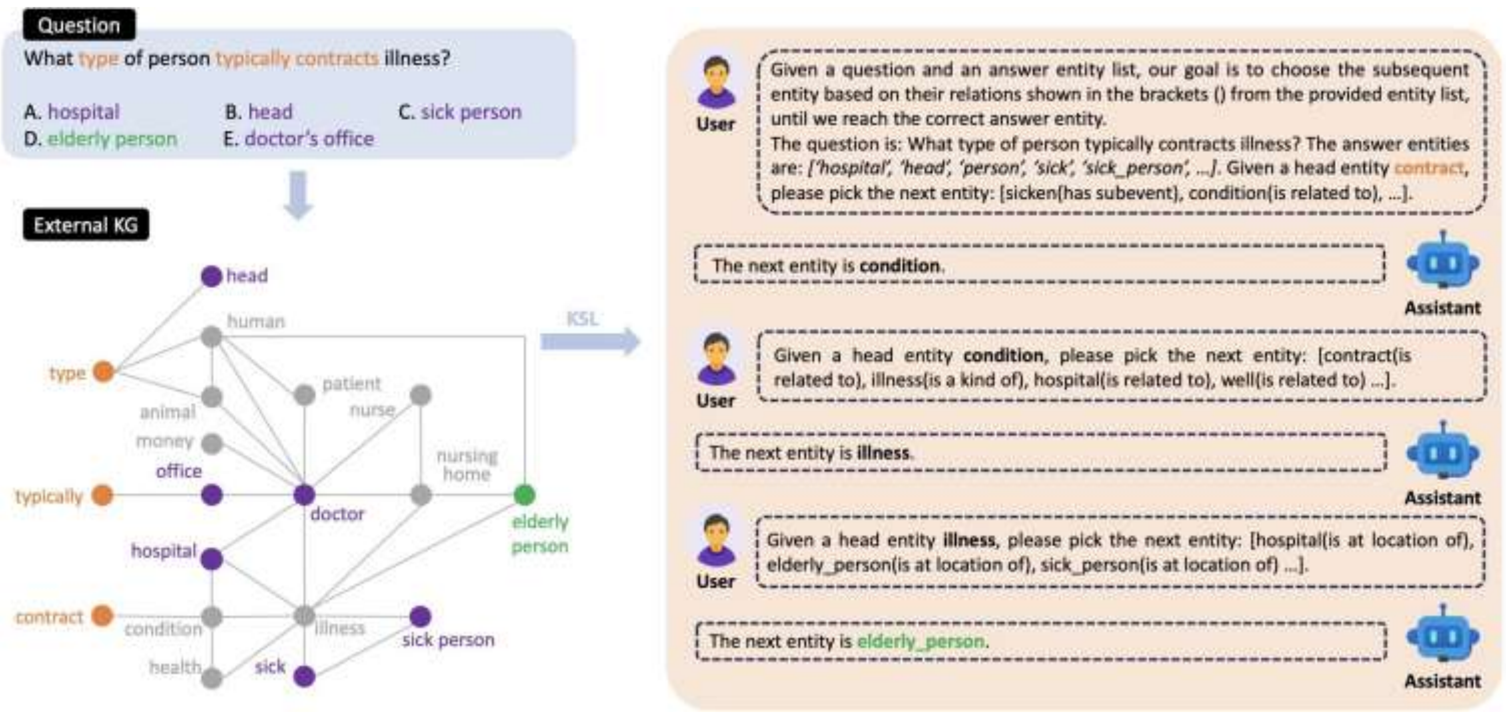
- **Knowledge Graph-augmented Language Models for Complex Question Answering (KGQA)**
  - The facts from the KG were weighted/ ranked by a KGQA before being fed into the LLM



LLMs Empowered by KGs

- **Knowledge Solver**: teaches LLMs to traverse KGs in a multi-hop way to reason the answer to a question.

- For given question answer choices pair [q, A], encode G_sub into text prompt TK to inject knowledge into LLMs.
- G_sub contains all nodes on the k-hop paths between nodes in Vq and Va.

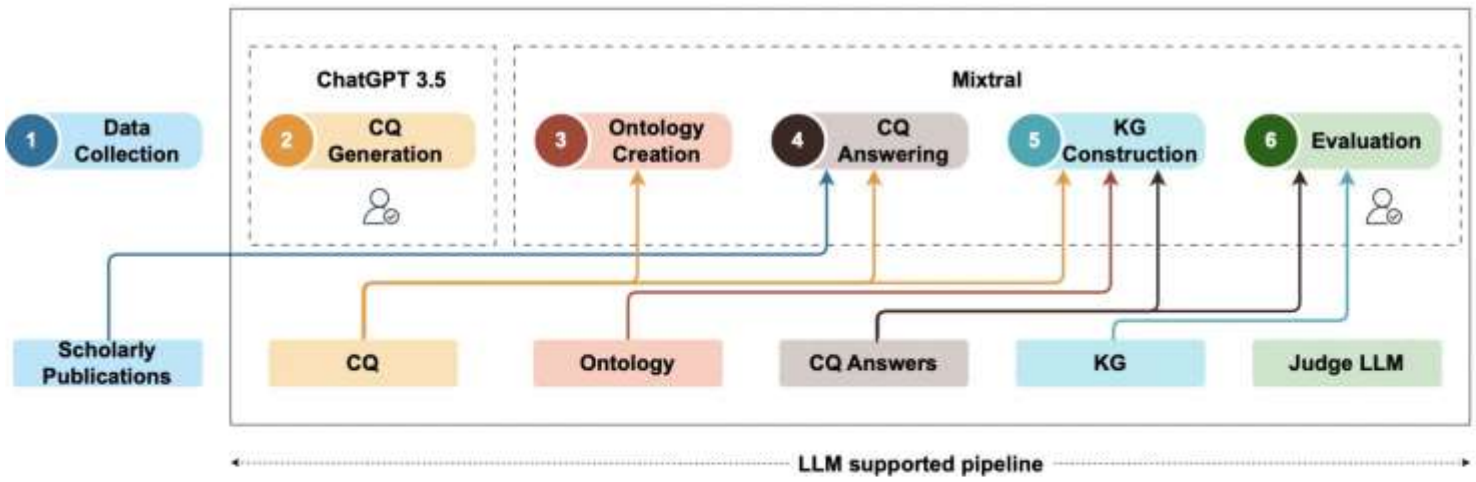


KGs Empowered by LLMs

Knowledge graph construction.

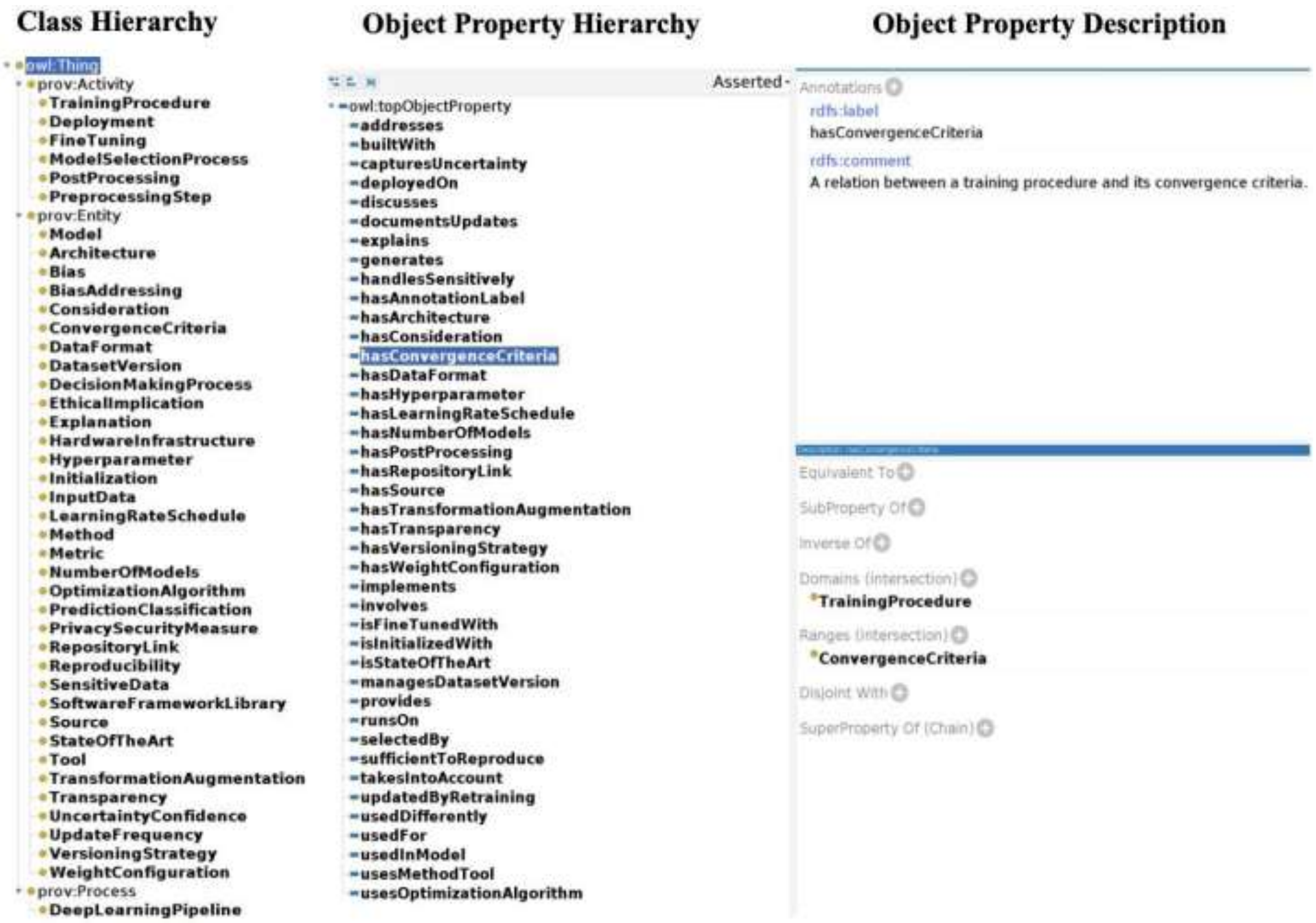
- time-consuming and costly construction process of KGs

An LLM supported approach to *ontology* and *knowledge graph* construction



- **CQ (competency questions), generation:** prompt ChatGPT-3.5 to get abstract-level questions
- **Extract all concepts and their relationships** from the CQs.
- **Constructed an ontology** for describing information using an in-context example containing a basic ontology structure
- **CQ Answering**: retrieved answers for all the CQs
- **KG generation**: prompt the LLM to extract key entities, relationships, and concepts from the answers and map them onto the ontology to generate the KG.

KGs Empowered by LLMs

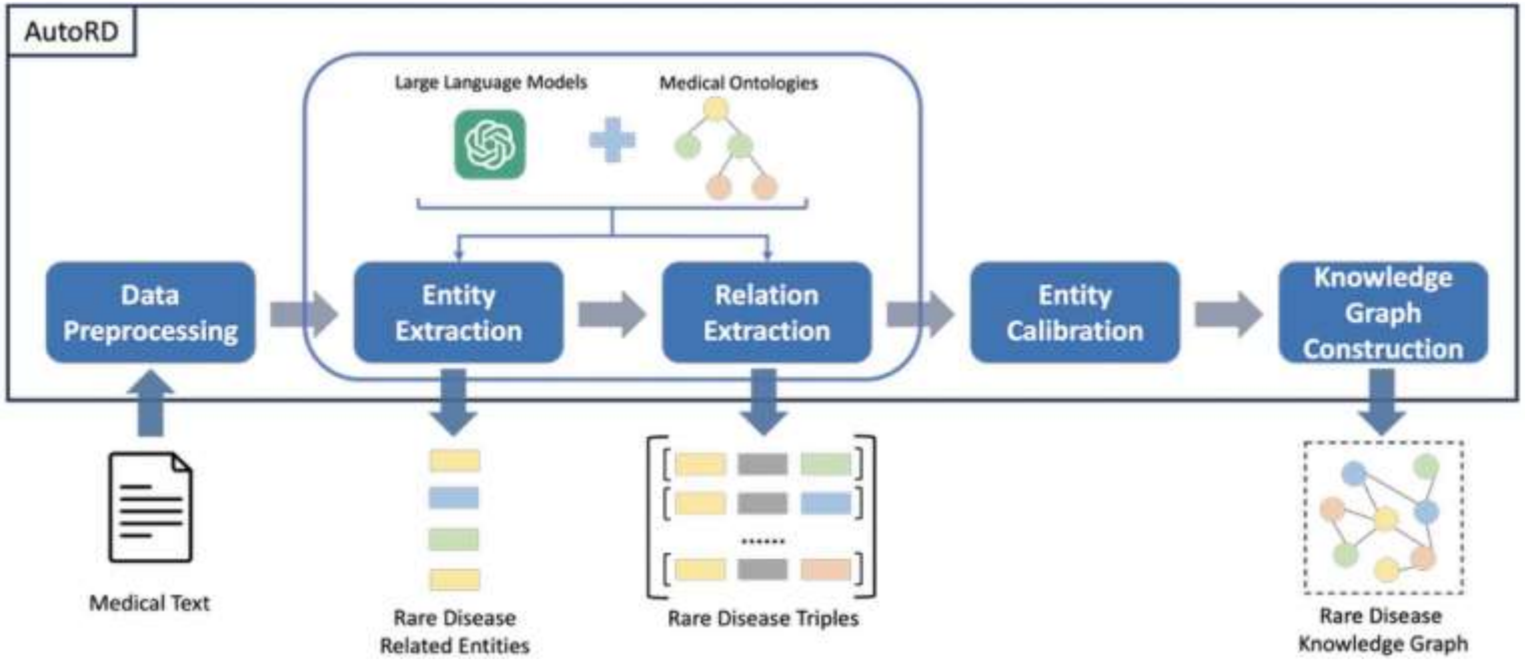


DLProv ontology generated by the LLM

Provenance is information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness.

KGs Empowered by LLMs

AutoRD: An Automatic and End-to-End System for Rare Disease Knowledge Graph Construction Based on Ontology-enhanced Large Language Models



KGs Empowered by LLMs

Extract More Terms

```
### Task:
(Extract entities: "anaphor", "medical_term"...)

### Definition:
## Entity:
(Define entities, including "rare_disease",
"disease", "symptom_and_sign", "anaphor",
"medical_term".)

### Notice:
(Remind the model of some considerations...)

(Provide extracted "medical_terms".)
{medical_terms}

### Input Text:
(Provide text.)
{text}

### Output format
## Entity Representation:
(Define entity representation.)

## Your output should be a single JSON object in
the following format:
(Define the format of response.)

### Output:
```

Extract Entities

```
### Task:
(Extract entities: "rare_disease", "disease",
"symptom_and_sign", "anaphor"...)

### Definition:
## Entity:
(Define entities, including "rare_disease",
"disease", "symptom_and_sign", "anaphor".)

### Notice:
(Remind the model of some considerations...)

### Examples
Here are some examples of input text and target
entities:
{examples}

### Input (Give Passage): {text}

### Extracted Medical Terms: {medical_terms}
### Extracted Anaphor: {anaphor}
### Rare Disease Knowledge:
{rare_disease_knowledge}

### Output format
## Entity Representation:
(Define entity representation.)

## Your output should be a single JSON object in
the following format:
(Define the format of response.)

### Output:
```

Extract Relations

```
### Task:
(Extract relations: "produces",
"increases_risk_of", "is_a", "is_acron",
"is_synon", "anaphora"...)

### Definition:
## Entity:
(Define entities, including "rare_disease",
"disease", "symptom_and_sign", "anaphor".)
## Relation:
(Define relations, including "produces",
"increases_risk_of", "is_a", "is_acron",
"is_synon", "anaphora".)

### Notice:
(Remind the model of some considerations...)

### Examples
Here are some examples of input text and target
entities:
{examples}

### Input (Give Passage): {text}

### Extracted Entities: {entities}

### Rare Disease Knowledge:
{rare_disease_knowledge}

### Output format
## Entity Representation:
(Define entity representation.)

## Relation Representation:
(Define relation representation.)

## Your output should be a single JSON object in
(Define the format of response.)

### Output:
```

Calibrate Entities

```
### Task:
(Verify all extracted entities and relations...)

### Definition:
## Entity:
(Define entities, including "rare_disease",
"disease", "symptom_and_sign", "anaphor".)
## Relation:
(Define relations, including "produces",
"increases_risk_of", "is_a", "is_acron",
"is_synon", "anaphora".)

### Notice:
(Remind the model of some considerations...)

### Input (Give Passage): {text}

### Extracted Entities: {entities}
### Extracted Relations: {relations}

### Output format
## Entity Representation:
(Define entity representation.)

## Relation Representation:
(Define relation representation.)

## Your output should be a single JSON object in
the following format:
(Define the format of response.)

### Output:
```

This figure presents the simplified content of all prompts. The black text represents the original text of the instructions. Grey text indicates a summary of each part of the instructions. Blue text highlights the prompt slots, where external information and inputs can be inserted.

KGs Empowered by LLMs

Automated Construction of Theme-specific Knowledge Graphs

Problem description: Given a specific theme and a set of documents D with each document $d \in$ D describing relevant content about the theme, our task aims to extract the theme-related knowledge triples from D

Example ("EV battery" Theme KG construction). Given the theme "EV-battery" and the following text:

"Deep cycle batteries are used to provide continuous electricity to run electric vehicles like forklifts", the output of theme-specific knowledge graph construction may include the following possible knowledge triples: *(deep cycle batteries, provide, continuous electricity), (deep cycle batteries, be power source of, electric vehicles), (deep cycle batteries, be power source of, forklifts), (electric vehicles, include, forklifts).*

KGs Empowered by LLMs