Lecture 4: Reasoning

TIES4520 Semantic Technologies for Developers
Autumn 2018

University of Jyväskylä

Khriyenko Oleksiy

Reasoning types

- Two basic types:
  - Rule-based reasoning
    - General rule-based inference (semantic rules)
    - Further classification: forward-chaining and backward-chaining
  - Ontology-based reasoning
    - Classification-based inference (e.g. RDF-S, OWL reasoning)
    - The inference rules for RDF-S or OWL are ***fixed.*** Therefore: No need for rule engine -> procedural algorithm sufficient

```
:John :hasWife :Mary
```

+

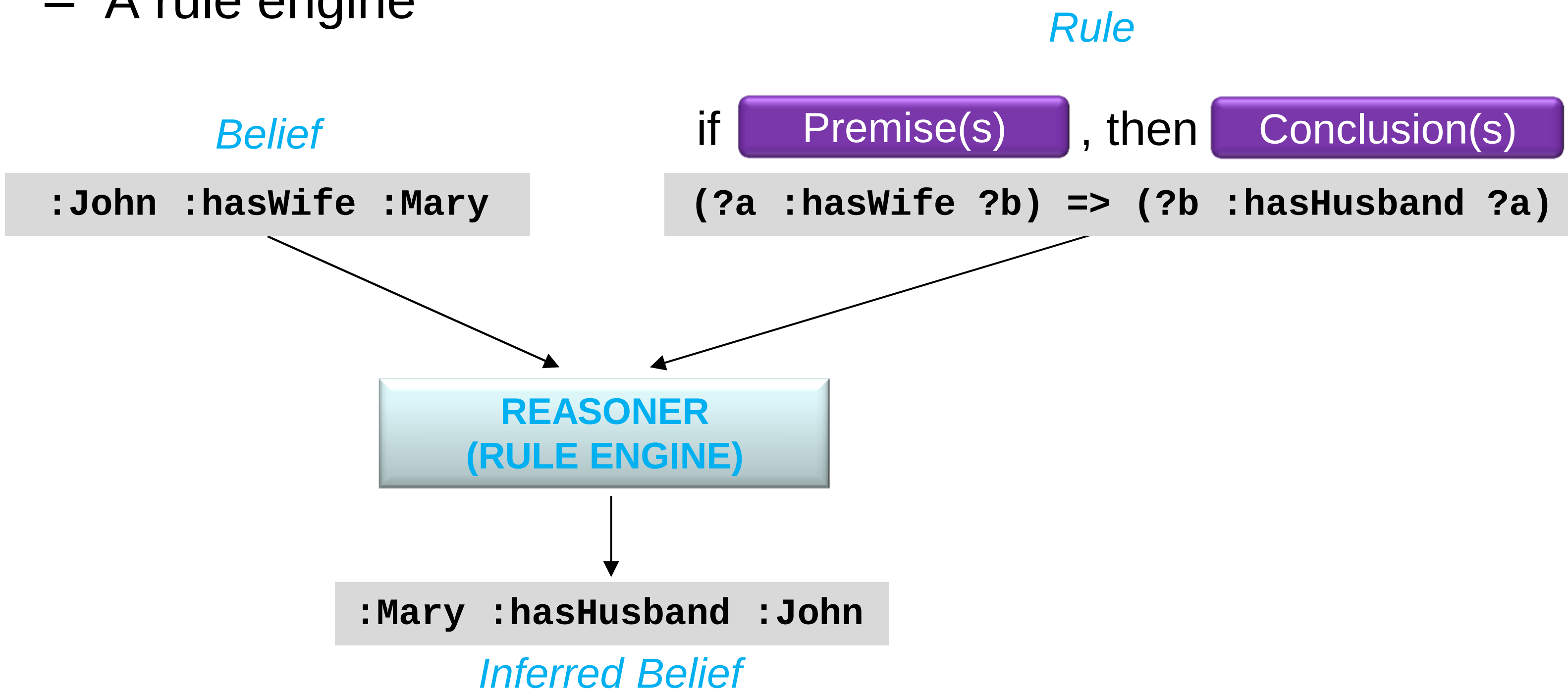
Family ontology

also means

```
:John rdf:type :Human .
:John rdf:type :Man .
:Mary rdf:type :Human .
:Mary rdf:type :Woman .
:Mary :hasHusband :John.
```

Rule-based reasoning

- The inference based on free-form rules always requires:
  - A language for representing the rules
  - A rule engine



Rule-based reasoning

- The OWL language is not able to express all relations (ex: it cannot express the relation “*child of married parents*“).
- The expressivity of OWL can be extended by adding rules to an ontology.
- Rule definition language:
  - ***SWRL*** (Semantic Web Rule Language)
  - ***Notation 3 (N3)*** logic
  - ***RIF*** (Rule Interchange Format)

SWRL

- ***SWRL*** (Semantic Web Rule Language):
  - Part of many tools (e.g. Hermit, Pellet, etc.)
  - Basic form is XML, but also available in human-readable form
  - unary predicates for describing classes and data types,
  - binary predicates for properties,
  - some special built-in n-ary predicates.
- SWRL rules are supported:
  - Protege OWL editor.
  - reasoners Pellet and Hermit.

SWRL

- SWRL predicates:
  - Class expressions: Class atom: `Person(?x)` `Man(Fred)`

```
Man(?p) -> Person(?p)
```

  - Property expressions:
  - Individual Property atom: `hasBrother(?x,?y)` `hasSibling(Fred,?y)`
  - Data Valued Property atom: `hasAge(?x,?age)` `hasAge(?x,232)` `hasName(Fred,”Fred”)`

```
Person(?p), hasSibling(?p,?s), Man(?s) -> hasBrother(?p,?s)
```

  - Data range restrictions

```
Person(?p), integer[>= 18,<= 65](?age), hasAge(?p, ?age) -> hasDriverAge(?p, true)
```

  - OWL Class expressions in SWRL Rules

```
Person(?x), hasChild min 1 Person(?x) -> Parent(?x)
```

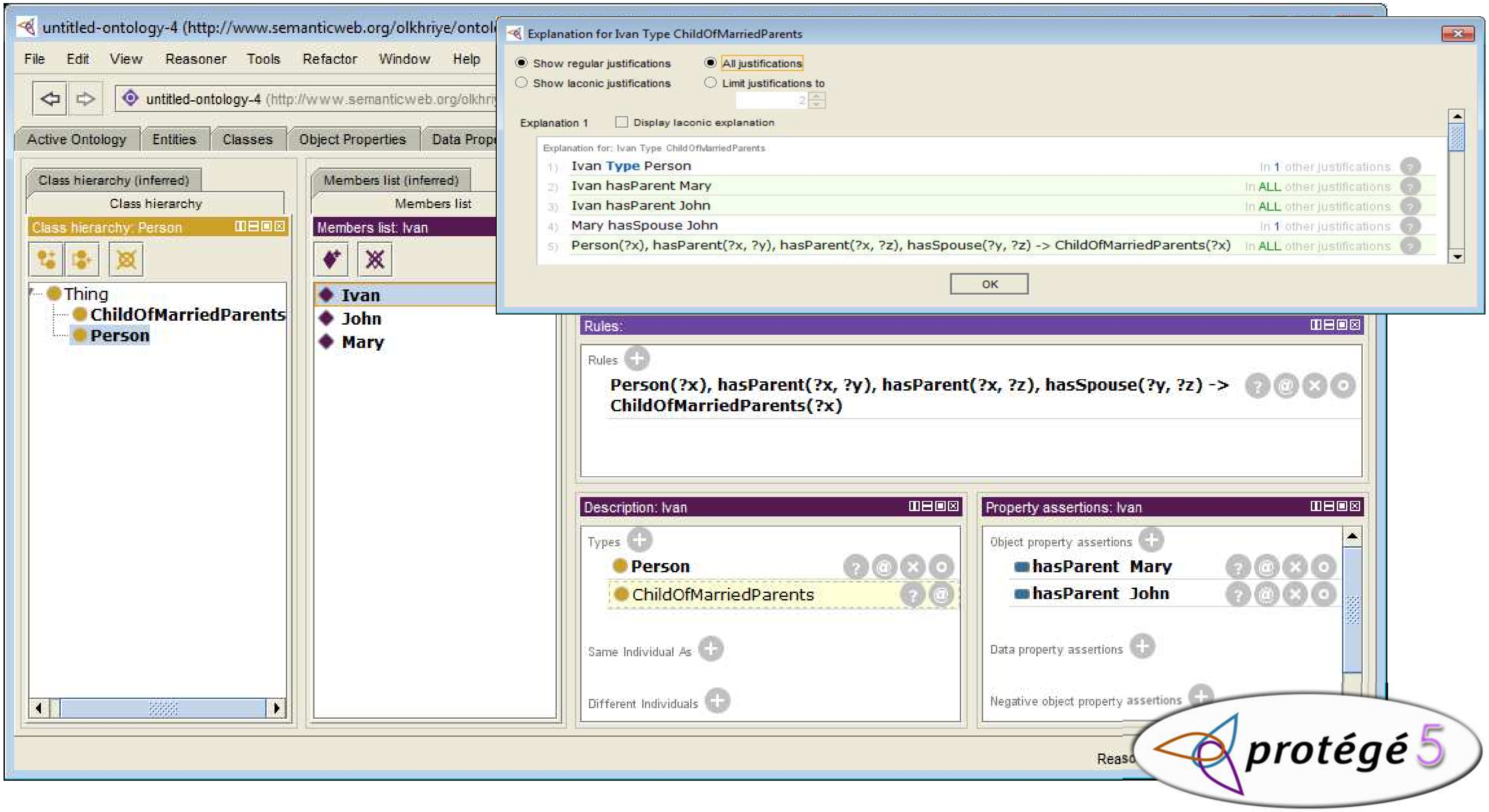
  - Core SWRL built-ins (*http://www.daml.org/rules/proposal/builtins.html*)

```
Person(?p), hasAge(?p, ?age), swrlb:greaterThan(?age, 18) -> Adult(?p)
```

```
Person(?p), bornOnDate(?p, ?date), xsd:date(?date),
swrlb:date(?date, ?year, ?month, ?day, ?timezone) -> bornInYear(?p, ?year)
```

SWRL

- Rule definition in Protégé 5.x (4.x)
  - Open rule tub from the menu: *Window – Views – Ontology Views – Rules*



Notation 3 (N3) logic rules

- *Notation 3 (N3)* logic rule expression
  - graph definition ({}) give a possibility to write formulas in rules

```
@prefix log: <http://www.w3.org/2000/10/swap/log#>.
@prefix family: <http://www.myOntology.org/family/>.
@forAll :x, :y, :z.
{ :x family:parent :y . :y family:brother :z } log:implies { :x family:uncle :z }.
```

- Shorthand symbols:
  - *?* – for universal variables "*@forAll*";
  - *_:* or better *[ ]* – for existential variables "*@forSome*" (blank node);
  - **=>** – for implies (*log:implies*);
  - **<=>** – for meaning (*log:means*);
  - **=** – for equivalents (*owl:equivalentTo*);

```
{?x family:parent ?y. ?y family:brother ?z} => {?x family:uncle ?z}.
```

- Built-in Functions: used by CWM (*http://www.w3.org/2000/10/swap/doc/CwmBuiltins*)

```
{ ?x f:age ?ag . ?ag math:lessThan 30} => { ?x rdf:type f:YoungPerson } .
```

```
{ ex:d test:point ?x.  ?x math:sin ?y } =>  {...} .
```

```
{ ex:testData ex:value ?x .
    ( ?x 1 ) math:sum ?y.
    (  ?y  " is one more than " ?x ) string:concatenation ?s
} => { ex:result ex:value ?s }.
```

RIF

- ***RIF*** (Rule Interchange Format) is W3C Recommendation June 2010

RIF is part of the infrastructure for the semantic web. The design of RIF is based on the observation that there are many "rules languages" in existence, and what is needed is to exchange rules between them.

- RIF includes three dialects:
  - Core dialect (which is extended into others)

```
Document(  Prefix(ppl <http://example.com/people#>)
           Prefix(cpt <http://example.com/concepts#>)
           Prefix(bks <http://example.com/books#>)
  Group(   Forall ?Buyer ?Item ?Seller (
                  cpt:buy(?Buyer ?Item ?Seller) :- cpt:sell(?Seller ?Item ?Buyer))
                  cpt:sell(ppl:John bks:LeRif ppl:Mary)))
```

  - Basic Logic Dialect (BLD)

```
Document(  Base(<http://example.com/people#>)
           Prefix(cpt <http://example.com/concepts#>)
           Prefix(bks <http://example.com/books#>)
  Group(   Forall ?Buyer ?Item ?Seller (
                  cpt:buy(?Buyer ?Item ?Seller) :- cpt:sell(?Seller ?Item ?Buyer))
                  cpt:sell(<John> bks:LeRif "Mary"^^rif:iri)))
```

  - Production Rule Dialect (PRD)

```
Prefix(ex <http://example.com/2008/prd1#>)
(* ex:rule_1 *)
Forall ?customer ?purchasesYTD (
 If   And( ?customer#ex:Customer
           ?customer[ex:purchasesYTD->?purchasesYTD]
           External(pred:numeric-greater-than(?purchasesYTD 5000)) )
 Then Do( Modify(?customer[ex:status->"Gold"]) ) )
```

Some rules of RDF Schema

- If a resource is an instance of a class, it is also an instance of any super-class of that class (*any human is a mammal*).

```
:Mammal rdf:type owl:Class.
:Human rdf:type owl:Class.
:Human rdfs:subClassOf :Mammal.
:John rdf:type :Human.
```

also means

```
:John rdf:type :Mammal.
```

```
{ ?A rdfs:subClassOf ?B. ?S rdf:type ?A } => { ?S rdf:type ?B } .
```

- If a statement with a property is made, the statement with any super-property is also true (*if you love something, you also like it*).

```
:like rdf:type owl:ObjectProperty.
:love rdf:type owl:ObjectProperty.
:love rdfs:subPropertyOf :like.
:John :love :Mary.
```

also means

```
:John :like :Mary.
```

```
{ ?P rdfs:subPropertyOf ?R. ?S ?P ?O } => { ?S ?R ?O } .
```

Some rules of RDF Schema

- Having defined *domain* and *range* of a property, we may conclude that:
    - o the resource, which has this property, belongs to the class associated with the *domain* of the property;
    - o the resource, which is referred as a value of the property, belongs to the class associated with the *range* of the property.

```
:Man rdf:type owl:Class.
:Woman rdf:type owl:Class.
:hasWife rdf:type owl:ObjectProperty;
       rdf:domain :Man;
       rdf:range :Woman.

:John :hasWife :Mary.
```

also means

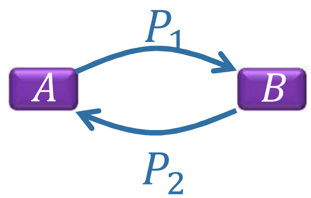
```
:John rdf:type :Man.
:Mary rdf:type :Woman.
```

```
{ ?P rdfs:domain ?C. ?S ?P ?O } => { ?S rdf:type ?C } .

{ ?P rdfs:range ?C. ?S ?P ?O } => { ?O rdf:type ?C } .
```

Some rules of OWL

- Some of the property characteristics allow reasoners to infer new knowledge about instances and their relations:
  - *owl:inverseOf*




```
:Human rdf:type owl:Class .
:hasChild rdf:type owl:ObjectProperty .
:hasParent rdf:type owl:ObjectProperty .
:hasChild owl:inverseOf :hasParent .
:John rdf:type :Human .
:Mary rdf:type :Human .
:John :hasChild :Mary .
```

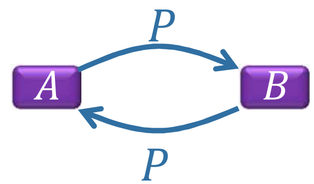
also means

```
:Mary :hasParent :John .
```

```
{ ?P owl:inverseOf ?Q . ?S ?P ?O } => { ?O ?Q ?S } .
```

Some rules of OWL

- Some of the property characteristics allow reasoners to infer new knowledge about instances and their relations:
  - *owl:SymmetricProperty*




```
:Human rdf:type owl:Class .
owl:inverseOf rdf:type owl:SymmetricProperty .
:hasChild rdf:type owl:ObjectProperty .
:hasParent rdf:type owl:ObjectProperty .
:hasChild owl:inverseOf :hasParent .
:John rdf:type :Human .
:Mary rdf:type :Human .
:Mary :hasParent :John .
```

```
{ ?P rdf:type owl:SymmetricProperty. ?S ?P ?O } => { ?O ?P ?S } .
```

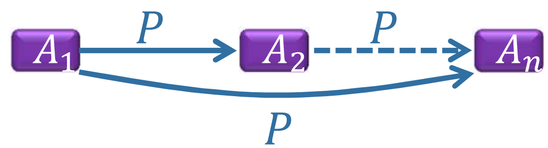
```
:hasParent owl:inverseOf :hasChild .
```

```
{ ?P owl:inverseOf ?Q . ?S ?P ?O } => { ?O ?Q ?S } .
```

```
:John :hasChild :Mary .
```

Some rules of OWL

- Some of the property characteristics allow reasoners to infer new knowledge about instances and their relations:
  - *owl:TransitiveProperty*



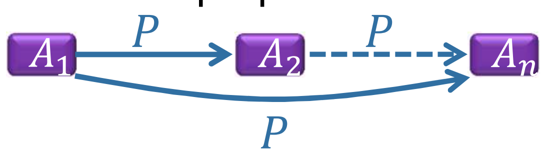

```
:Human rdf:type owl:Class .
:bossOf rdf:type owl:TransitiveProperty .
:John rdf:type :Human .
:Michael rdf:type :Human .
:Mary rdf:type :Human .
:John :bossOf :Mary .
:Mary :bossOf :Michael .
```

```
{ ?P rdf:type owl:TransitiveProperty. ?S ?P ?X. ?X ?P ?O } => { ?S ?P ?O } .
```

```
:John :bossOf :Michael .
```

Some rules of RDF Schema

- Transitive property: If class A is a sub-class of B, while B is a sub-class of C, then A is a sub-class of C (*mother is woman, woman is human, therefore mother is human*). Also applies to sub-properties
- Example: *rdfs:subClassOf* and *rdfs:subPropertyOf* are transitive properties




```
:Human rdf:type owl:Class.
:Woman rdf:type owl:Class.
:Mother rdf:type owl:Class.
:Woman rdfs:subClassOf :Human.
:Mother rdfs:subClassOf :Woman.
:prefer rdf:type owl:ObjectProperty.
:like rdf:type owl:ObjectProperty.
:love rdf:type owl:ObjectProperty.
:like rdfs:subPropertyOf :prefer.
:love rdfs:subPropertyOf :like.
```

also means

```
:Mother rdfs:subClassOf :Human.
:love rdfs:subPropertyOf :prefer.
```

```
{ ?B rdfs:subClassOf ?C. ?A rdfs:subClassOf ?B }=>{ ?A rdfs:subClassOf ?C } .
{ ?Q rdfs:subPropertyOf ?R. ?P rdfs:subPropertyOf ?Q }=>{ ?P rdfs:subPropertyOf ?R } .
```

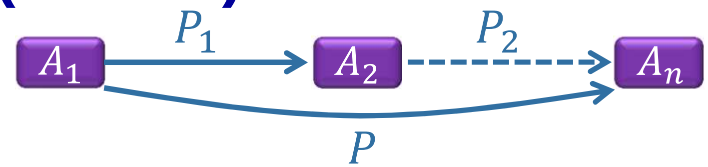
```
:Mary rdf:type :Mother.
:John rdf:type :Human.
:Mary :love :John.
```

also means

```
:Mary rdf:type :Woman.
:Mary rdf:type :Human.
:Mary :like :John.
:Mary :prefer :John.
```

```
{?A rdfs:subClassOf ?B. ?S rdf:type ?A}=>{?S rdf:type ?B}.
{ ?P rdfs:subPropertyOf ?R. ?S ?P ?O } => { ?S ?R ?O } .
```

Property chains (OWL-2)



- *owl:propertyChainAxiom* ***(OWL-2)***

  Simply: If the property P_1 relates individual A_1 to individual A_2, and property P_2 relates individual A_2 to individual A_n, then property P relates individual A_1 to individual A_n;

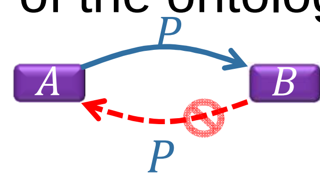
```
:hasParent rdf:type owl:ObjectProperty .
:hasGrandparent rdf:type owl:ObjectProperty ;
                owl:propertyChainAxiom ( :hasParent :hasParent ) .
:hasGrandGrandparent rdf:type owl:ObjectProperty .
[ rdf:type owl:ObjectProperty ;
  owl:propertyChainAxiom (:hasGrandparent :hasParent)] rdfs:subPropertyOf :hasGrandGrandparent.
[ rdf:type owl:ObjectProperty ;
  owl:propertyChainAxiom (:hasParent :hasGrandparent)] rdfs:subPropertyOf :hasGrandGrandparent.
:Human rdf:type owl:Class .
:John rdf:type :Human .
:Michael rdf:type :Human .
:Mary rdf:type :Human .
:Katarina rdf:type :Human ;
          :hasParent :Mary .
:Mary :hasParent :Michael .
:Michael :hasParent :John .
```

```
:Katarina :hasGrandparent :Michael .
:Mary :hasGrandparent :John .
```

```
:Katarina :hasGrandGrandparent :John .
```

Some rules of OWL

- Some of the property characteristics set certain conditions and allow reasoners to detect inconsistency of the ontology:
  - *owl:AsymmetricProperty*




```
:Human rdf:type owl:Class .
:isChildOf rdf:type owl:AsymmetricProperty .
:John rdf:type :Human .
:Mary rdf:type :Human .
:John :isChildOf :Mary .
…
:Mary :isChildOf :John .   inconsistency
```

  - *owl:IrreflexiveProperty*


```
:Human rdf:type owl:Class .
:motherOf rdf:type owl:IrreflexiveProperty .
:John rdf:type :Human .
:Mary rdf:type :Human .
:Mary :motherOf :John .
…
:Mary :motherOf :Mary .   inconsistency
```

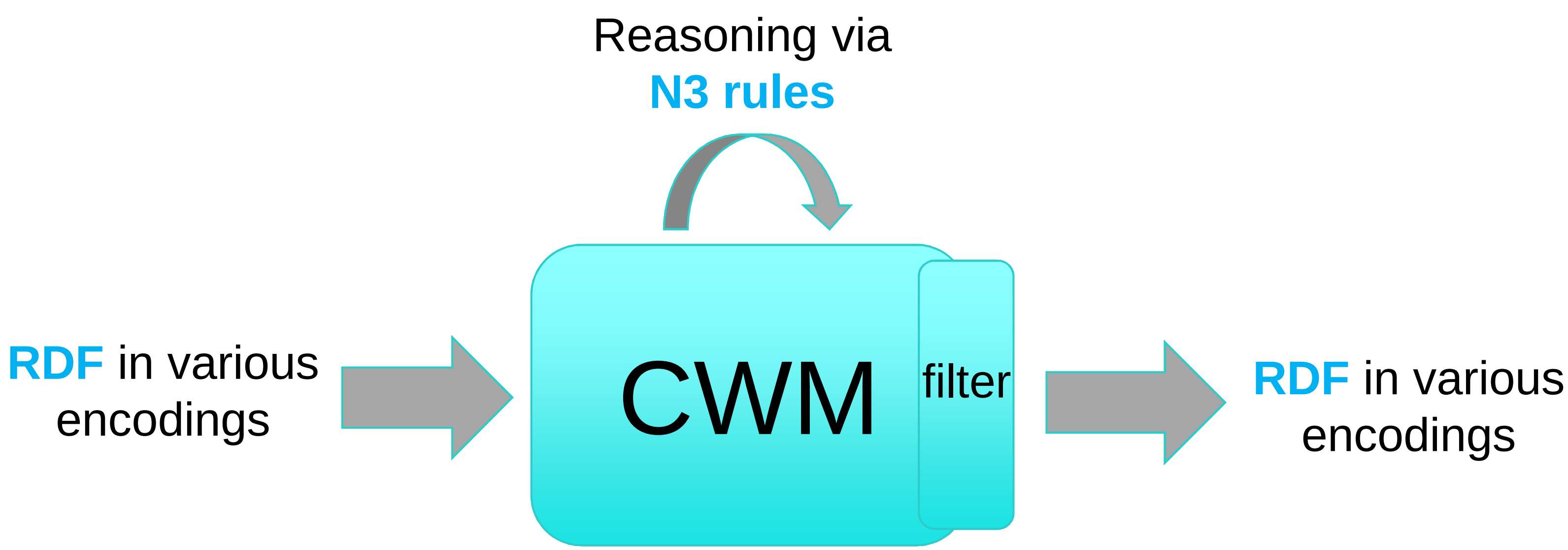
Forward vs. backward-chaining reasoning

- Forward
  - Input: rules + data
  - Output: extended data
  - Starts with available facts
  - Uses rules to derive new facts (which can be stored)
  - Stops when there is nothing else to be derived
- Backward
  - Input: rules + data + hypothesis (statement)
  - Output: Statement is true / Statement is false
  - Goes backwards from the hypothesis to the set of axioms (our data)
  - If it can find the path to the original axioms, then the hypothesis is true (otherwise false)

CWM

- ***Forward-chaining reasoner*** written in Python
- Originally to show capabilities of N3
- Link: *http://www.w3.org/2000/10/swap/doc/cwm.html*
- ”Cwm (*pronounced coom*) is a general-purpose data processor for the semantic web”. It can be used for:
  - *querying,*
  - *checking,*
  - *transforming,*
  - *filtering information…*
- Deals with open worlds!
- CWM’s function:
  - Loads data in N3 or RDF/XML + rules in N3
  - Applies rules to data
  - Output result in N3 or RDF/XML

CWM


Reasoning via
N3 rules
RDF in various encodings
CWM
filter
RDF in various encodings

CWM usage

practical

- You must have command *python* in your PATH variable

Use Python v2.7 (not v3.5)

```
e.g. set PATH=%PATH%;c:\Python27
```

- Basic usage:

```
python cwm <COMMAND> <OPTIONS> <STEPS>
```

- By default the output goes to standard output
  - If you want to store it in a file, use redirect, e.g.:

```
python cwm input.n3 --think --data --rdf > result.rdf
```

- Useful use cases:

source format *source file* *destination format*

```
python cwm --n3 data.n3 --filter=rules.n3 --n3
```

Show only new reasoned facts by applying rules in rules.n3

```
python cwm --n3 data.n3 --apply=rules.n3 --n3
```

Show both the old data from data.n3 together with new reasoned data

```
python cwm --n3 data.n3 --think=rules.n3 --n3
```

As –apply, but continue until no more rule matches (or forever!)

CWM usage: Example

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://www.example.org/someExample#> .
```

Data

```
:Human rdf:type owl:Class .
:dan rdf:type :Human .
:peter rdf:type :Human .
:mary rdf:type :Human .
:jon rdf:type :Human .
:betty rdf:type :Human .

:ancestorOf rdf:type owl:TransitiveProperty.
:hasSpouse rdf:type owl:SymmetricProperty.
:brotherOf rdf:type owl:ObjectProperty.
:sisterOf rdf:type owl:ObjectProperty.
owl:inverseOf rdf:type owl:SymmetricProperty .

:brotherOf owl:inverseOf :sisterOf .

:dan :ancestorOf :peter .
:peter :ancestorOf :jon .
:peter :hasSpouse :mary .
:betty :sisterOf :jon .
```

+

Rules

```
{ ?P rdf:type owl:SymmetricProperty .
  ?S ?P ?O
} => {?O ?P ?S} .

{ ?P owl:inverseOf ?Q .
  ?S ?P ?O
} => {?O ?Q ?S} .

{ ?P rdf:type owl:TransitiveProperty .
  ?S ?P ?X .
  ?X ?P ?O
} => {?S ?P ?O} .
```

```
:dan :ancestorOf :jon .
:mary :hasSpouse :peter .
:sisterOf owl:inverseOf :brotherOf .
```

```
:jon :brotherOf :betty .
```

CWM practical tips

practical

- When you write a file for CWM, *always put dot after the last statement!*
- You don't have to separate data and rules into two files
  - If you use N3 as your notation, then they can be in one file
  - Example:
    ```
    python cwm --n3 dataAndRules.n3 --rules
    python cwm --n3 dataAndRules.n3 --think
    ```
- CWM can be used to convert files without reasoning
  - Example:
    ```
    python cwm --rdf source.rdf --n3 > destination.n3
    ```
- More CWM command line arguments are available at (*http://www.w3.org/2000/10/swap/doc/CwmHelp*) or using `python cwm --help`

SPARQL Rules (SPIN)

SPARQL Rules are a collection of RDF vocabularies such as the ***SPARQL Inferencing Notation (SPIN)*** that build on the W3C SPARQL standard. These vocabularies let you define new functions, stored procedures, constraint checking, and inferencing rules for your Semantic Web models, and all these definitions are stored using object-oriented conventions and the RDF and SPARQL standards. *http://spinrdf.org/ , http://spinrdf.org/spinsquare.html*

SPIN makes it possible to attach executable rules to classes. Rules are represented as ***SPARQL CONSTRUCT*** queries that apply to all instances of the associated class and its subclasses. In those rules, the variable ***?this*** refers to each instance of those classes.

```
ss:Rectangle spin:rule [
   rdf:type sp:Construct ;
   sp:text """
      CONSTRUCT {
         ?this ss:area ?area .                    # Infer ?area as a value of ss:area
      }
      WHERE {
         ?this ss:width ?width .                  # Get the width of ?this Rectangle
         ?this ss:height ?height .                # Get the height of ?this Rectangle
         BIND ((?width * ?height) AS ?area) .     # Compute area := width * height
      }
   """ ;
] .
```

Getting Started with SPARQL Rules (SPIN): *http://www.topquadrant.com/technology/sparql-rules-spin/*
http://www.topquadrant.com/spin/tutorial/

GraphDB reasoning

Rulesets is a sets of axiomatic triples, consistency checks and entailment rules, which determine the applied semantics. A ruleset file has three sections named *Prefices*, *Axioms*, and *Rules*.

http://graphdb.ontotext.com/documentation/standard/reasoning.html

- Prefixes defines the abbreviations for the namespaces used in the rest of the file.

```
Prefices
{
    rdf  :  http://www.w3.org/1999/02/22-rdf-syntax-ns#
    rdfs :  http://www.w3.org/2000/01/rdf-schema#
    owl  :  http://www.w3.org/2002/07/owl#
    xsd  :  http://www.w3.org/2001/XMLSchema#
}
```

- Axioms asserts axiomatic triples, which usually describe the meta-level primitives used for defining the schema such as *rdf:type*, *rdfs:Class*, etc. It contains a list of the (variable free) triples, one per line.

```
Axioms
{
    // RDF axiomatic triples
    <rdf:type>      <rdf:type> <rdf:Property>
    <rdf:subject>   <rdf:type> <rdf:Property>
    <rdf:predicate> <rdf:type> <rdf:Property>
    <rdf:object>    <rdf:type> <rdf:Property>
    <rdf:first>     <rdf:type> <rdf:Property>
    <rdf:rest>      <rdf:type> <rdf:Property>
    <rdf:value>     <rdf:type> <rdf:Property>
    <rdf:nil>       <rdf:type> <rdf:List>
}
```

GraphDB reasoning

- Entailment rules

```
Id: <rule_name>
    <premises> <optional_constraints>
    -------------------------------
    <consequences> <optional_constraints>
```

```
Rules
{
Id: rdf1_rdfs4a_4b
    x  a  y
    -------------------------------
    x  <rdf:type>  <rdfs:Resource>
    a  <rdf:type>  <rdfs:Resource>
    y  <rdf:type>  <rdfs:Resource>

Id: rdfs2
    x  a  y                [Constraint a != <rdf:type>]
    a  <rdfs:domain>  z    [Constraint z != <rdfs:Resource>]
    -------------------------------
    x  <rdf:type>  z

Id: owl_FunctProp
    p  <rdf:type>  <owl:FunctionalProperty>
    x  p  y       [Constraint y != z, p != <rdf:type>]
    x  p  z       [Constraint z != y] [Cut]
    -------------------------------
    y  <owl:sameAs>  z
}
```

GraphDB reasoning

- Consistency checks. You can define rulesets that contain consistency rules. When creating a new repository, set the check-for-inconsistencies configuration parameter to ***true***. It is *false* by default (for compatibility with the previous OWLIM releases). The syntax is similar to that of rules, except that Consistency replaces the Id tag that introduces normal rules. Also, consistency checks do not have any consequences and indicate an inconsistency whenever their premises can be satisfied, e.g.:

```
Consistency: something_can_not_be_nothing
    x rdf:type owl:Nothing
    -------------------------------


Consistency: both_sameAs_and_differentFrom_is_forbidden
    x owl:sameAs y
    x owl:differentFrom y
    -------------------------------
```

Predefined rulesets The pre-defined rulesets provided with GraphDB cover various well-known knowledge representation formalisms and are layered in such a way that each one extends the preceding one.

empty	No reasoning, i.e., GraphDB operates as a plain RDF store.
rdfs	Supports the standard model-theoretic RDFS semantics.
owl-horst	OWL dialect close to OWL Horst - essentially pD*
owl-max	RDFS and that part of OWL Lite that can be captured in rules (deriving functional and inverse functional properties, all-different, subclass by union/enumeration; min/max cardinality constraints, etc.).
owl2-ql	The OWL2 QL profile - a fragment of OWL2 Full designed so that sound and complete query answering is LOGSPACE with respect to the size of the data. This OWL2 profile is based on DL-LiteR, a variant of DL-Lite that does not require the unique name assumption.
owl2-rl	The OWL2 RL profile - an expressive fragment of OWL2 Full that is amenable for implementation on rule engines.

GraphDB reasoning

- Custom rulesets. GraphDB has an internal rule compiler that can be configured with a custom set of inference rules and axioms. You may define a custom ruleset in a ***.pie*** file (e.g., MySemantics.pie). The easiest way to create a custom ruleset is to start modifying one of the .pie files that were used to build the precompiled rulesets.

All examples below use the sys: namespace, defined as:

```
prefix sys: <http://www.ontotext.com/owlim/system#>
```

Add a custom ruleset from *.pie* file

```
INSERT DATA {
    _:b sys:addRuleset <file:c:/graphdb/test-data/test.pie>
}
```

```
INSERT DATA {
    <:custom> sys:addRuleset <http://example.com/test-data/test.pie>
}
```

```
INSERT DATA {
    _:b sys:addRuleset "owl-max"
}
```

```
INSERT DATA {
    <:custom> sys:addRuleset
        '''Prefixes { … }
           Axioms { … }
           Rules { … }'''
}
```

Reinferring. If reconnected to a repository with a different ruleset, it does not take effect immediately. However, you can cause reinference with:

```
INSERT DATA { [] <http://www.ontotext.com/owlim/system#reinfer> [] }
```

Euler/EYE

- Originally ***backward-chaining reasoner*** for N3 logic – inference engine ***Euler***
- ***Euler YAP Engine (EYE)*** – a backward-forward-backward chaining reasoner design enhanced with Euler path detection (reasoning is grounded in First Order Logic).
- Home: *http://www.agfa.com/w3c/euler/ , https://github.com/josd/eye*
- Download: *http://sourceforge.net/projects/eulersharp/files/eulersharp/*
  https://github.com/josd/eye
- Implemented in several languages: *Java, C#, Python, Javascript and Prolog*
- Input: *rules + data + hypothesis*
- Output: *Chain of rules that lead to the hypothesis (if the hypothesis is true)*

Other reasoners

- Racer by Racer Systems (open-source)

http://www.ifis.uni-luebeck.de/~moeller/racer/

- Jena inference support (open-source)

http://jena.apache.org/

- Pellet: OWL DL reasoner for Java (open-source)

https://github.com/complexible/pellet

- FaCT++ (open-source, in C++)

http://code.google.com/p/factplusplus/

- JFact DL Reasoner: a Java port of theFaCT++ (open-source)

http://jfact.sourceforge.net/

- HermiT Owl Reasoner (open-source)

http://hermit-reasoner.com/

- RDF4J(Sesame) supports RDFS reasoning
- RDF4J(Sesame) supports a forward-chaining SPIN rule engine (currently in beta)

SQWRL

SQWRL (**S**emantic **Q**uery-Enhanced **W**eb **R**ule **L**anguage; pronounced *squirrel*) is a *SWRL*-based query language that provides SQL-like operators for extracting information from OWL ontologies.

https://github.com/protegeproject/swrlapi/wiki/SQWRL

The language provides two sets of query operators:

- Core Operators (*https://github.com/protegeproject/swrlapi/wiki/SQWRLCore*)
- Collection Operators (*https://github.com/protegeproject/swrlapi/wiki/SQWRLCollections*)

Running SQWRL Queries. Two mechanisms are provided by the SWRLAPI to execute SQWRL queries:

- a Java API that provides a JDBC-like interface, called the SQWRL Query API, which can be used to execute queries and retrieve query results in Java applications.
  https://github.com/protegeproject/swrlapi/wiki/SQWRLQueryAPI
- a graphical user interface called the SQWRL Query Tab that supports interactive querying and results display. The SQWRL Query Tab is available in both the *Protégé SWRLTab Plugin* and the *standalone SWRLTab*. *https://github.com/protegeproject/swrlapi/wiki/SQWRLQueryTab*

SQWRL

SWRL core examples:

```
Adult(?p) -> sqwrl:select(?p)
```

```
Person(?p) ^ hasAge(?p, ?a) ^ swrlb:lessThan(?a, 25) -> sqwrl:select(?p, ?a)
```

- Counting: *sqwrl:count, sqwrl:countDistinct*

```
Person(?p) ^ hasName(?p, ?name) -> sqwrl:countDistinct(?name)
```

- Aggregation. Basic aggregation is supported four operators: *sqwrl:min, sqwrl:max, sqwrl:sum, sqwrl:avg.*

Any numeric variable ***not passed*** to a *sqwrl:select* operator can be aggregated…

```
Person(?p) ^ hasAge(?p, ?age) -> sqwrl:avg(?age)
```

- Grouping

```
Person(?p) ^ hasDrug(?p,?d) ^ hasDose(?p,?dose)-> sqwrl:select(?p,?d) ^ sqwrl:avg(?dose)
```

- Ordering of Results. Ordered using the *sqwrl:orderBy* and *sqwrl:orderByDescending* operators.

```
Person(?p) ^ hasName(p, ?name) ^ hasCar(?p, ?c)
  -> sqwrl:select(?name) ^ sqwrl:count(?c) ^ sqwrl:orderBy(?name)
```

- Selecting a Subset of Results: *sqwrl:limit, sqwrl:firstN, sqwrl:lastN, sqwrl:notFirstN, sqwrl:notLastN, sqwrl:leastN, sqwrl:greatestN, sqwrl:notLeastN, sqwrl:notGreatestN, sqwrl:nth, sqwrl:nthLast, sqwrl:notNth, sqwrl:notNthLast, sqwrl:nthSlice, sqwrl:nthLastSlice*, etc.

```
Person(?p) ^ hasName(?p, ?name) -> sqwrl:select(?name) ^ sqwrl:limit(2)
```

- Result Columns

```
Person(?p) ^ hasName(?p,?namer) ^ hasCar(?p,?c) -> sqwrl:select(?name, "Number of cars")
^ sqwrl:count(?c) ^ sqwrl:columnNames("Name", "Description", "Count")
```

Task 4