Semantic Web
(Winter Term 2022/23)

(c) Prof Dr. Wolfgang May
Universität Göttingen, Germany

may@informatik.uni-goettingen.de

With contributions by Franz Schenk, Lars Runge, Sebastian Schrage
and indirectly several of our students.

Lecture: Advanced Course (Master) in Informatics; 3+1 hrs/week, 6 ECTS Credit Points
Lab Course: 2 hrs/week + exercises, 6 ECTS Credit Points

- participants should have knowledge of first-order logic syntax, semantics, model theory and tableau calculus.
- the slide set contains an introduction to first-order logic (e.g. taught in the modules “Formale Systeme” (in German, BSc) and “Deductive Databases”) for recalling concepts and introducing the notation used in this lecture.

A comprehensive German-English dictionary can e.g. be found at

`http://dict.leo.org/`

AIMS OF THE COURSE

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation." – Tim Berners-Lee, James Hendler, Ora Lassila, "The Semantic Web", in *Scientific American*, May 2001.

- What is an ontology?
- The "Semantic Web Tower"
- RDF as a data model, query languages etc.
- RDFS: a restricted language and model theory for schema metadata
- underlying theoretical concepts: Description Logics
- OWL

Chapter 1
Introduction

The Buzzword of the Day: Semantic Web

"invented" by Tim Berners-Lee in an article in *Scientific American*, May 2001.

- computer-understandable semantics of data
- more "intelligent" applications use this semantics
- integration of data
- integration of behavior

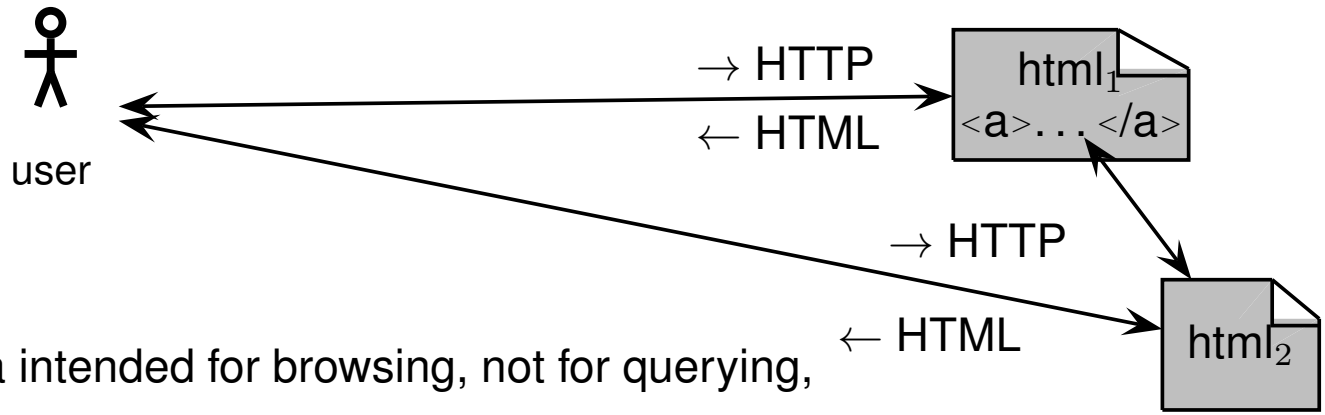
Different people mean quite different things when talking about the "Semantic Web" (e.g., focusing on "Semantic technologies", "Web 2.0", "Knowledge Representation").

This lecture collects topics that are also useful outside the "Semantic Web", and provide important foundations for the "Semantic Web".

1.1 Data on the Web

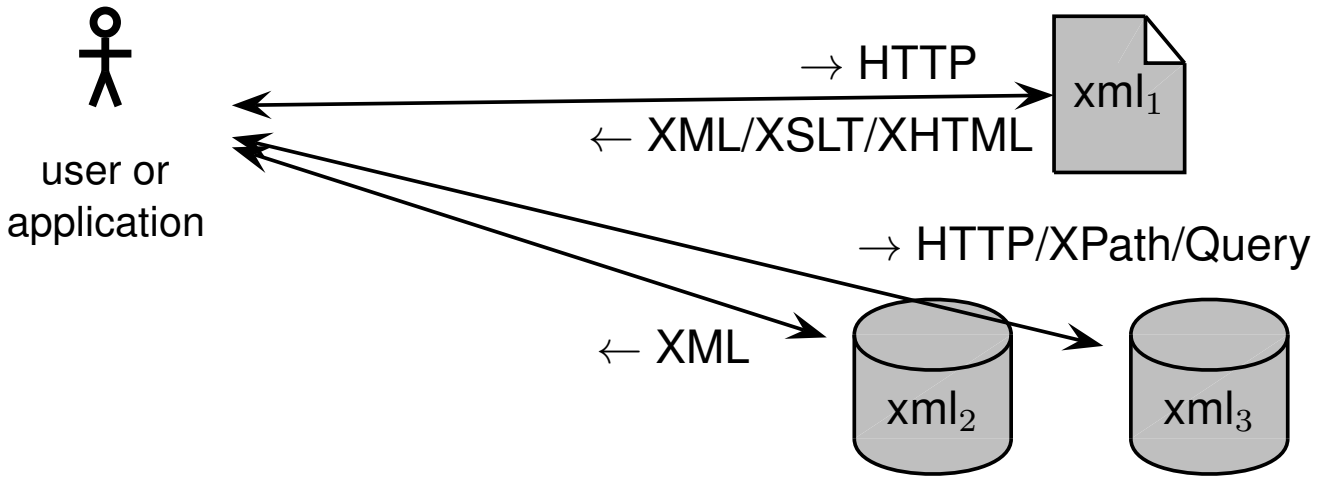
- World Wide Web:
  - Design of HTML started in 1989 (CERN), standard in 1991,
  - Browsing, Links between sources (“Web”)
- XML in the WWW:
  - XML as a data model + family of related languages.
  - XML data sources
  - Web-Services: successor of CORBA, based on XML data exchange
  - no “XML Web”, but mainly isolated sources, nearly not interlinked (XLink)
  - problem of *data integration* between sources
  - querying “the Web” as e.g. “give me a (train/flight/...) connection from Göttingen to St. Malo next weekend” not possible
  - users disappointed
- existing facts not appropriately accessible
- information = facts + semantics

ARCHITECTURE: THE HTML WEB



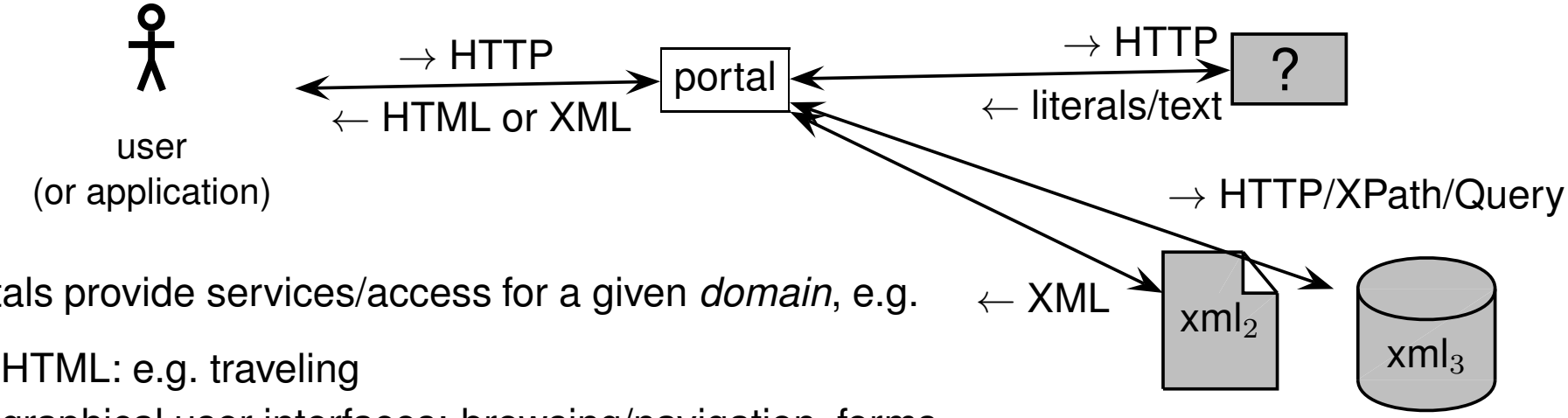
- HTML data intended for browsing, not for querying,
- access by absolute URL or by navigation through the Web,
- HTML (text, tables, list) is “data”, but still far from information, is “machine readable”, but only understandable for the browser, not for data-related tasks,
- “data integration” task mostly done offline (look up flight FRA-CDG and CDG-RNS at http://www.billigerfliegen.de, write down departure time, look up connection GOE-FRA at http://www.bahn.de),
- extracting information from HTML “data” requires detailed wrapper programming (into a common data model) based on the representational structure of the HTML pages. (example: generation of the Mondial database in 1997)

ARCHITECTURE: THE XML WEB



- XML is a data model
- can be wrapped by XSLT stylesheets to HTML for browsing,
- nearly no links between XML documents (XLink not used),
- XML data integration by high-level languages (XSLT, XQuery), requires knowledge of source DTDs or XML Schemas, manually written integration program (XQuery, XSLT).

XML Web: Data Integration with Portals

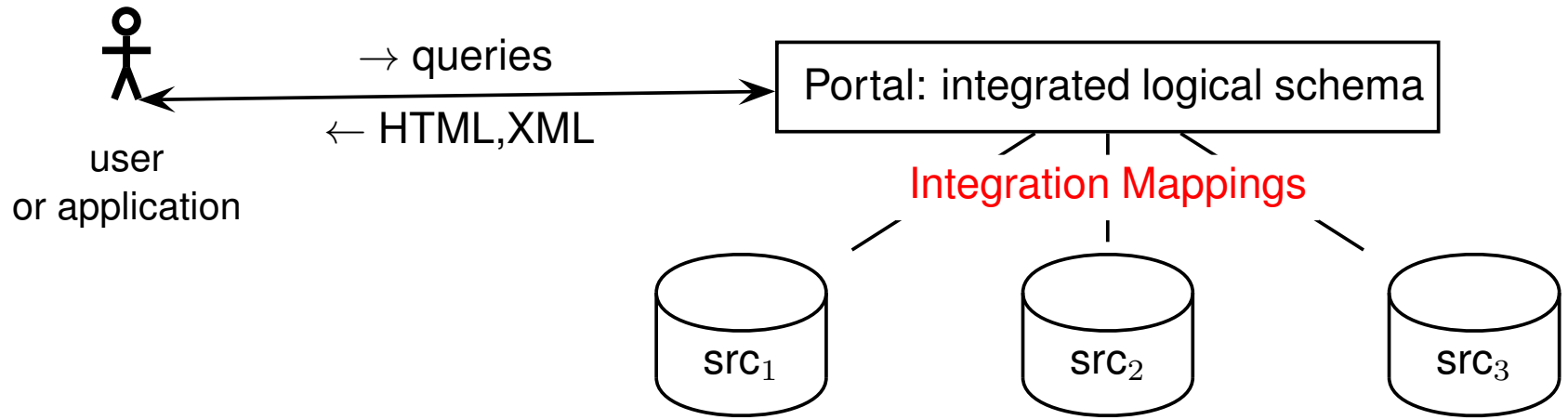


Portals provide services/access for a given *domain*, e.g.

- HTML: e.g. traveling
  graphical user interfaces: browsing/navigation, forms
  e.g. `http://www.billiger-fliegen.de` is a portal to airlines
- maps different schemas to the common one
  - wrappers for non-XML data sources (legacy),
  - (mostly hand-written) integration program in high-level language,
- obvious advantages if sources use a commonly agreed DTD/XML Schema,
- must be adapted if schema of a source changes or a source is removed or added.
- much more difficult: portal for access by computers
- computers lack the background knowledge that a user has wrt. a graphical interface

DATA INTEGRATION: CENTRALISED ARCHITECTURE

- federated databases of the 80s and 90s
- WWW portals (since late 90s)

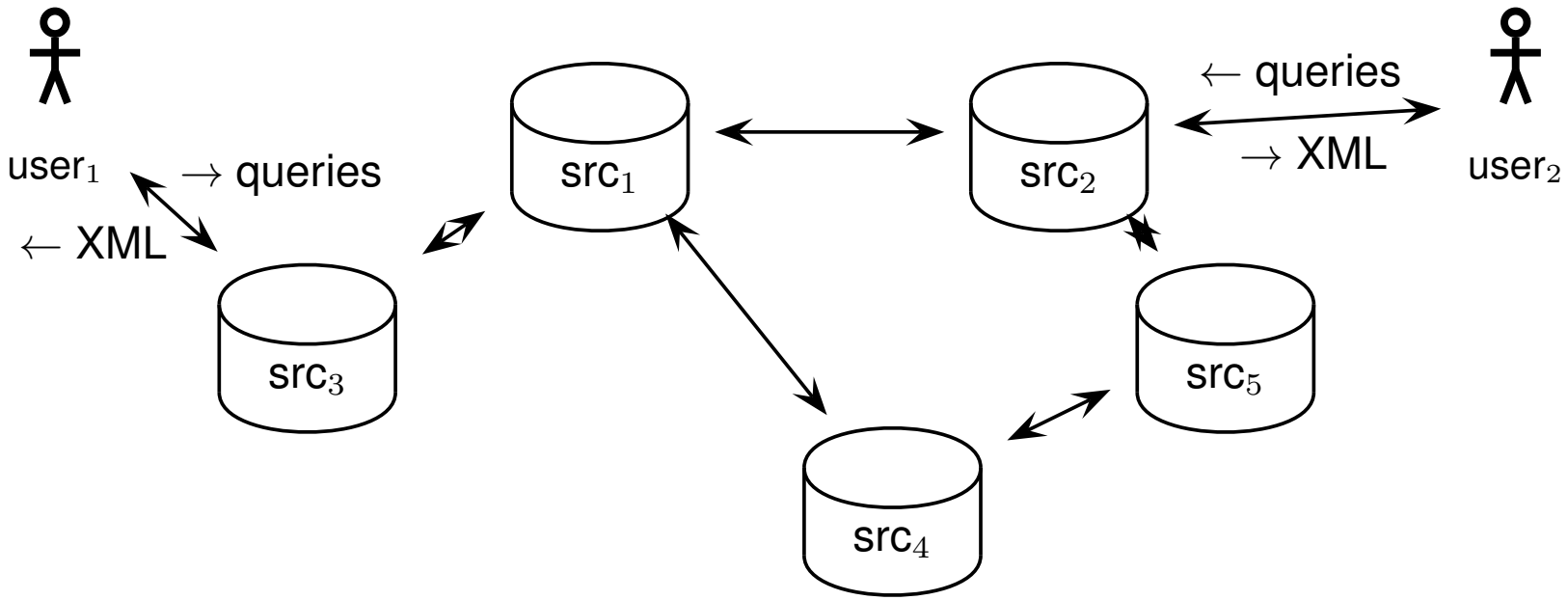


Integration Mapping

- usually “Global as View” (GAV), defining the integrated database as a view over the original sources,
- “Warehouse/materialized approach”: evaluate queries regularly and maintain an integrated database,
- “Virtual Approach”: queries are stated against the integrated schema and *rewritten* (in GAV: view expansion) and stated against the original databases.

DATA INTEGRATION: PEER-TO-PEER ARCHITECTURE

- P2P: since mid-90s (Tsimmis/OEM, Piazza, Napster, Gnutella, ...)



- communication between nodes
- Peer-to-Peer mappings between sources (bidirectional)
  worst case: exponential number of mappings needed
- located in the data nodes (e.g., using XLinks and some embedding)
- main issues in P2P: routing + integration mappings

1.2 Outlook: Semantic Web

Make data acquisition and integration easier *and* more powerful:

- underlying technology: Internet + XML, Web Services,
- agreed terminology ("ontology") throughout an application domain:
  - syntax and semantics of names,
  - agreed schema for identifiers of entities
    ($\Rightarrow$ matching of entities between different sources),
- unique mapping from the ontology to the data model,
- knowledge base instead of simple database:
  - expressive ontology language,
  - logical foundation, model theory,
  - reasoning mechanisms.
- semantics-based applications.

DATA MODEL AND REPRESENTATION: RDF

Goal: data model *independent* from representational structure and syntax.

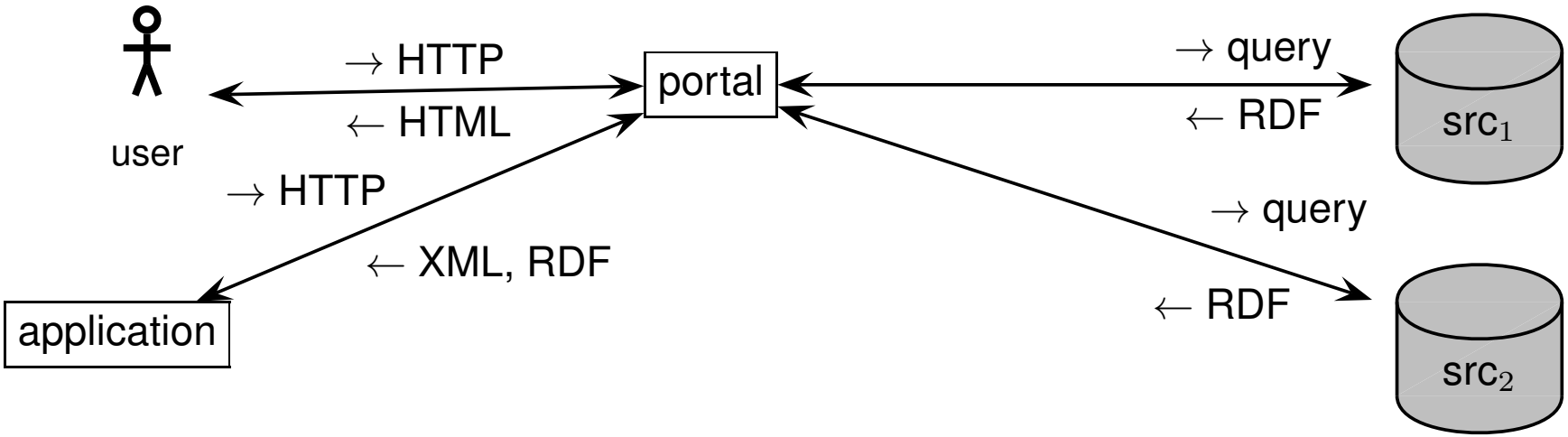
- *graph-based* data model
  - vertices: entities (identified by URIs – Unique Resource Identifiers) and values
  - edges: relationships and properties
  - all facts can be expressed by triples: Subject-Predicate-Object,
  - unified inherent semantics of the triples.
- metadata included within the data (RDFS, OWL) by predefined relationship names ("ontology vocabulary"),
- an ontology has a *unique* mapping to this data model,
- RDF data can be represented in XML
  - note: also relational data can be represented in XML – cf. SQLX
  - use XML *not* as a data model, but as a *data exchange format*.

ADDING "INTELLIGENCE" TO THE WEB

- Conceptual model of an ontology includes *knowledge*.
- reasoning not application-dependent, but based on generic concepts and formalisms that include schema, data, and further information:
  - class hierarchy
  - transitivity, inverse of properties, domains and ranges, cardinalities
  - derivation rules
  - algorithms/calculi
- logics, model-theory, model-based semantics of "the Web":
  "from (the information in) the Web one can infer . . . "
- reasoning (= application of Artificial Intelligence methods)

$\Rightarrow$ Description Logics, OWL (Web Ontology Language)

SEMANTIC WEB



- sources provide data in the RDF datamodel,
- transferred in a generic XML format,
- agreed names (ontology) and identifiers (URIs),
- data integration $\equiv$ union of facts + reasoning.
- In addition to this portal-based architecture, peer-2-peer communication is supported.

SEMANTIC WEB ARCHITECTURE

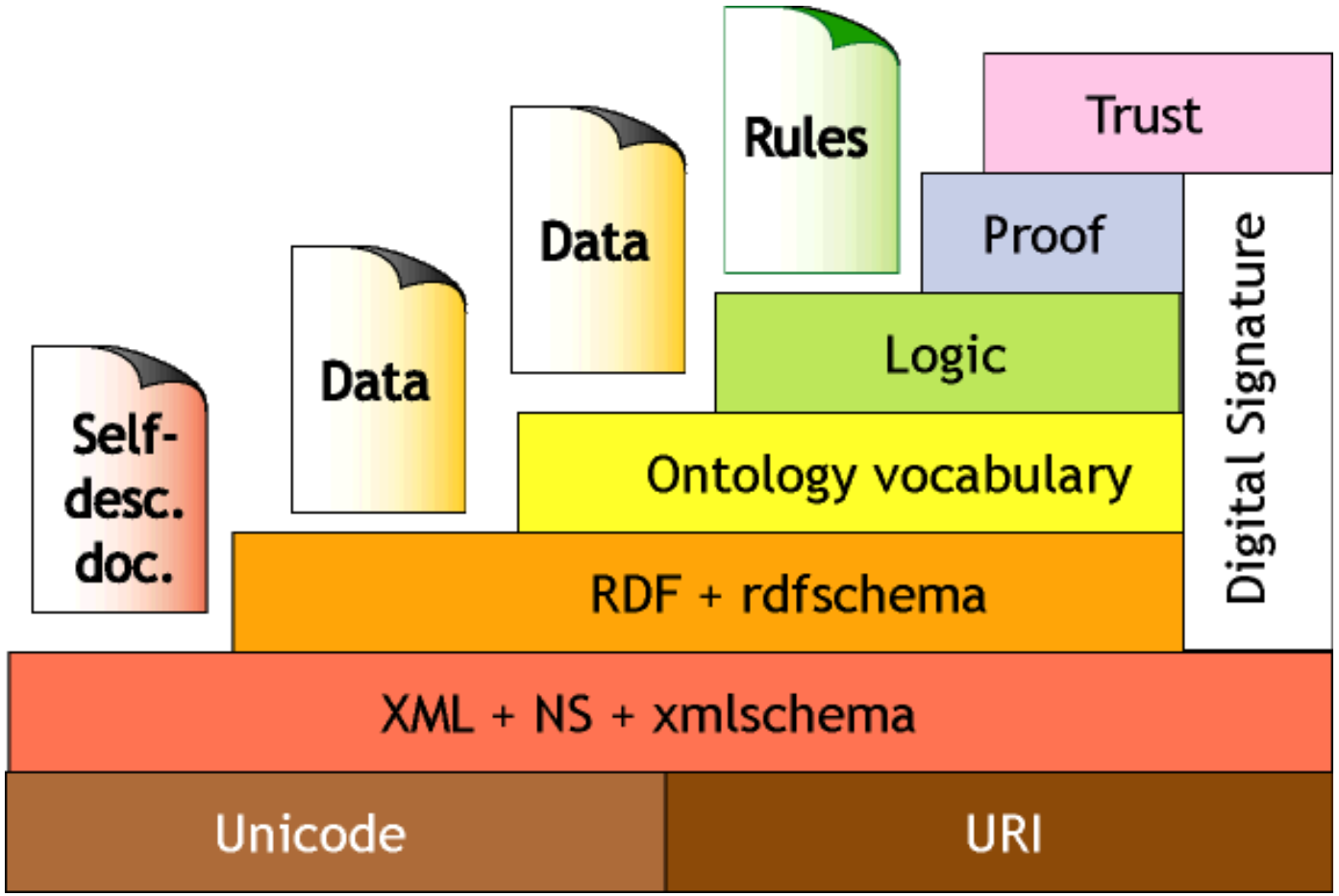
- intelligence located in application nodes and in the infrastructure
- Portals (integration + reasoning):
  serve e.g. as centralized entry points to a domain
- Peer-to-peer (P2P) communication:
  communication between “related” nodes

Additional Issues

- “find” information in the Semantic Web
  brokers & traders, yellow pages, registries, information propagation
  $\Rightarrow$ “Linked Open Data (LOD)”
- “social” aspects: communities of nodes, trust, recommender services.

THE VISION: SEMANTIC WEB TOWER

- Tim Berners-Lee, talk at a W3C meeting, 2000.



Chapter 2
Metadata and Ontologies

- metadata: data about data
  - classical in databases: schema
  - classical in documents: author and last-changed-date, keywords in header section of Web pages
    problem: which keywords to use?
- Ontology: describe and relate all kinds of facts and relationships that are relevant when talking about a *domain*
  - geography, bioinformatics, medicine, ...
  - biology: carnivores eat animals
- more than schema: schema with annotations *what* notions mean
- No consensus whether an ontology is only the metadata (schema level), or if it contains the metadata + the instances

ONTOLOGIES

Description of concepts

- what concepts
- properties
- relationships (hierarchy, uses-relationship, any others)

Simple kind of Ontology: Taxonomy

- taxonomy: hierarchy
- sometimes also: typical properties (base for taxonomy)

Example Taxonomy: Biology

- animals, plants, mammals, fishes, insects, carnivores, felidae, canidae, bovinae, ...
- with their identifying properties
- tasks:
  - given an instance, determine its most specific class
  - show/prove that certain things hold for all instances of a class

ONTOLOGIES: EXAMPLES

Academic Ontology

- concepts like lecturers, professors, PhD candidates, research groups, lectures, exams, publications, etc.,
- relationships such as “professors lead research groups”, “professors supervise PhD candidates” etc.
- Description Logics: “T-Box” (Terminology/Taxonomy, intensional knowledge)

The IFI Ontology

- also describes the instances, e.g., that DH, JG, WM and SW are the professors, SW is dean of studies, tmg and dbis are research groups, and WM leads the dbis group etc.
- Description Logics: “T-Box + A-Box” (assertional knowledge, extensional knowledge)

ONTOLOGY FORMALISMS IN INFORMATICS

- (a first-order logic language/signature is an ontology)
- an ER Model is an ontology
- an UML class diagram is an ontology

$\Rightarrow$ an agreed ontology allows for interoperation and data interchange

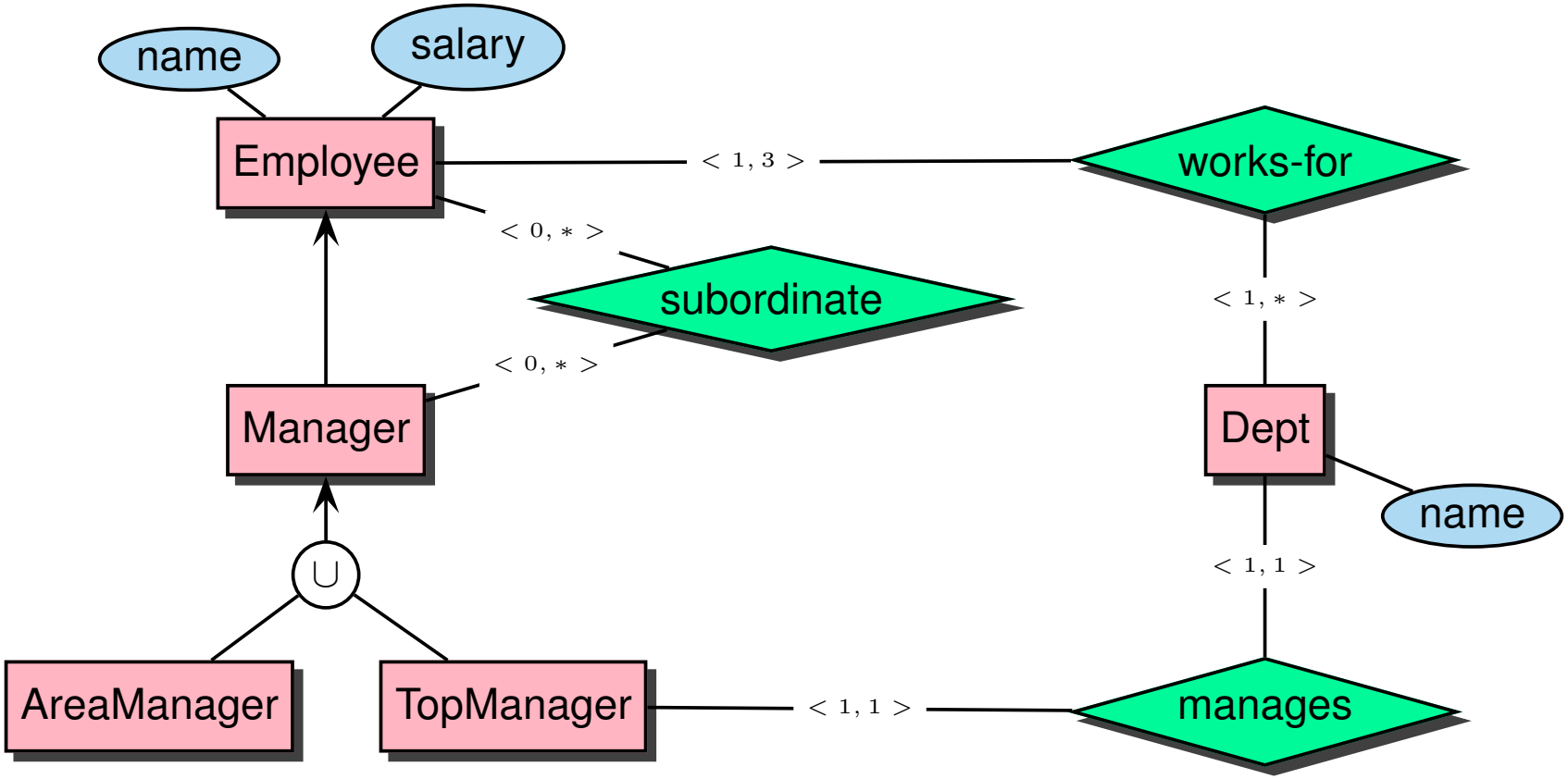
- an XML DTD or XML Schema is an ontology.
  It defines "names" *and* representational structure as an XML tree
  $\Rightarrow$ it is more restrictive as an ontology should be.

Goals

- ontology formalism wrt. an *abstract* data model
- integration not of (XML) data structures, but of (abstract) models, representable in ... XML
- for reasoning (proving consistency or deriving knowledge), an ontology formalism must be based on some kind of logic

First-order logic as an ontology formalism?
first-order logic is undecidable

2.1 Example: ER Diagram as an Ontology



(TopManager: leads a department;
AreaManager: intermediate group leaders in a department)

What can be “models” of this ontology? How do you represent them? Give an example.

Semantics: Set theory

- a class is a set of instances,
  Employee={alice, bob, john, mary, tom, larry} and Manager={alice, bob, john, mary},
  AreaManager={mary} and TopManager={alice, bob, john},
  Dept={sales, production, management}

  Constraints from subclasses:
  $Manager = AreaManager \cup TopManager$
  $Manager \subseteq Employee$
  $AreaManager \subseteq Manager$ and
  $TopManager \subseteq Manager$ (both redundant)

- an attribute is a set of pairs of (i) an instance and (ii) an element of a literal domain (constraint!)
  name =
  {(alice, "Alice"), (bob, "Bob"), (john, "John"), (mary, "Mary"), (tom, "Tom"), (larry, "Larry")}
  salary =
  {(alice, 70000), (bob, 60000), (john, 100000), (mary, 40000), (tom, 25000), (larry, 20000)}
  analogously for department names.

Semantics: Set theory (Cont'd)

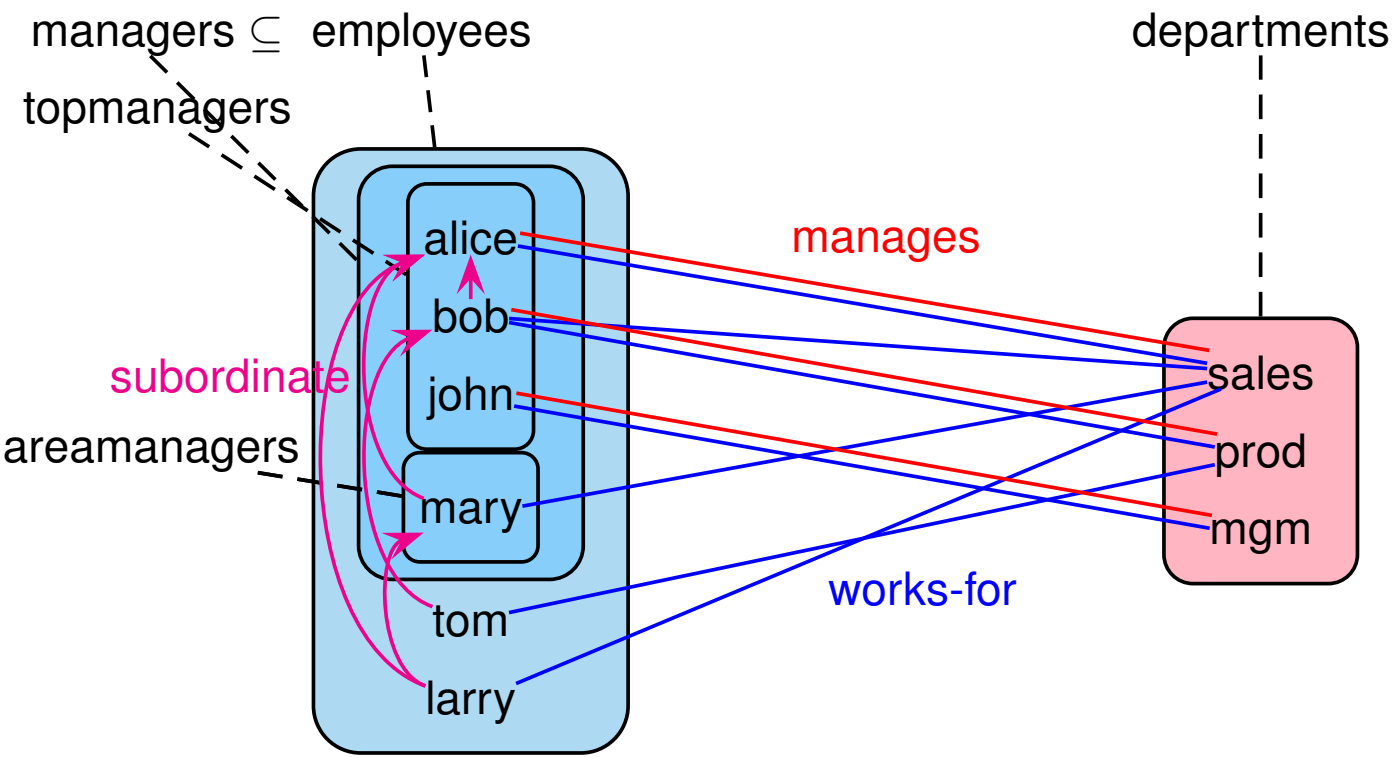
- a relationship is a set of pairs of instances,
  (or a set of n-tuples, in case of n-ary relationships)
  works-for $\subseteq$ *Employee* $\times$ *Dept*

  works-for = {(alice, sales), (mary, sales), (larry, sales), (bob, production),
  (bob, sales), (tom, production), (john, management)}

  manages $\subseteq$ *TopManager* $\times$ *Dept*
  manages = {(alice, sales), (bob, production), (john, management)}
  subordinate $\subseteq$ *Employee* $\times$ *Manager*
  subordinate = {(mary, alice), (bob, alice), (larry, mary), (larry, alice), (tom, bob)}

- not so obvious: constraints coming from the cardinality specifications, e.g.,
  the set of top managers is a subset of the things that manage exactly one department,
  the set of employees is a subset of the things that work for at least one and at most three departments
  (see later after the discussion of first-order logic)

Semantics: Set theory – Graphical Illustration



- does manages $\subseteq$ works-for hold in general?
- subordinate $\supseteq$ works-for $\circ$ manages^{-1} ?
- subordinate $\supseteq$ (works-for $\setminus$ manages) $\circ$ manages^{-1} !! ("$\setminus$" denotes set difference)
- the subordinates relationship wrt. area managers (larry$\rightarrow$mary) cannot be derived but must be stated explicitly.

Adequateness of (extended) ER Diagrams

An ontology should give a concise characterization of a domain and its constraints.

- classes, key constraints
- subclasses (specialization/generalization, disjointness)
- ranges and domains of properties (e.g., that the domain of “manages” is not all employees but only the managers)
- cardinalities
- is manages $\subset$ works-for ?
- is manages $\cap$ works-for = $\emptyset$?
- subproperty constraints cannot be expressed
- further constraints (e.g., employees that work for a department are subordinate to the department’s manager) cannot be expressed.

Exercise

- Discuss alternatives for the cardinalities for “subordinate”.

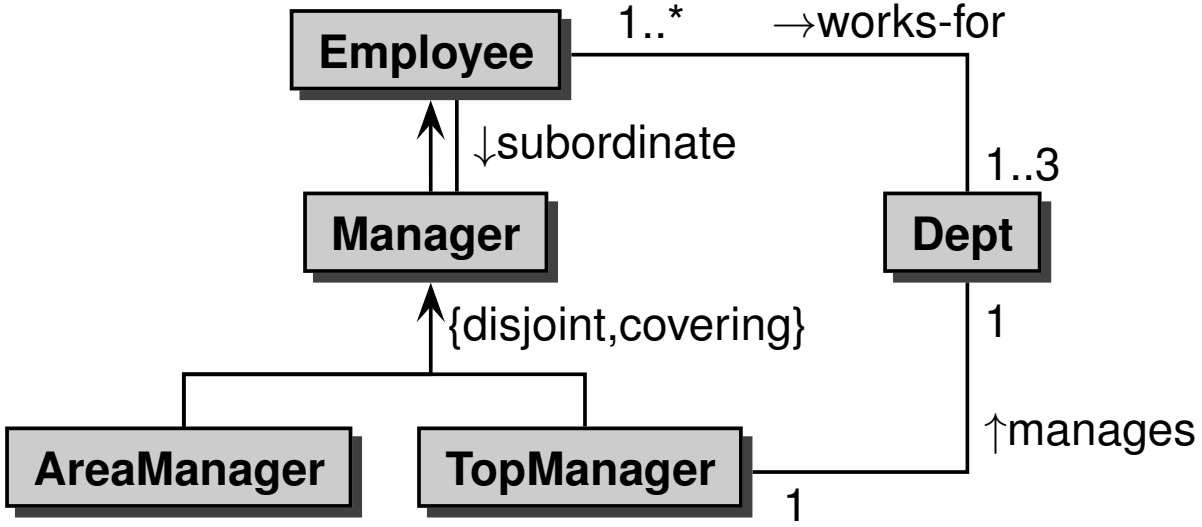
Alternative Semantics: Relational Model

Exercise: give a relational schema and the corresponding database state to the above ER diagram.

Relational Schema as an Ontology

- basically non-graphical, can be supported e.g. by dependency diagrams (cf. the Mondial documentation)
- no distinction between “classes” and “relationships”
- key constraints, foreign key/referential constraints
- keys/foreign key allow to guess classes vs. relationships
- cardinality “1”: functional dependencies (adding n:1-relationships into a table like department/manages country/capital)
- no general cardinality constraints
- sometimes: domain constraints (by foreign keys)
- no further (inter- and intra-relation) constraints

2.2 Example: UML Class Diagram as an Ontology



common: UML class diagrams + OCL (object constraint language) constraints

OCL: first-order logic formulas associated one or more with UML elements (e.g.,

subordinate $\supseteq \{(x, y) \mid \exists d$: works-for$(x, d) \wedge$ works-for$(y, d) \wedge$ manages$(y, d) \wedge x \neq y\}$

associated with associations subordinate, works-for and manages).

2.3 Summary and Outlook

Ontology Modeling Aspects

- an ontology describes the notions of a domain.
  (Note: sometimes it also describes the individuals)
- descriptions in database context (ER, UML) interpreted as *constraints*, but can also be read as *definitions* (Logical Rules, e.g., definition of “subordinate”).
- ER has a restricted expressiveness
- UML diagrams without OCL have a restricted expressiveness,
- UML with using OCL is as expressive as first-order logic.
- first-order logic: is too expressive (undecidable)

$\Rightarrow$ subsets of First-Order Logic as ontology formalisms

  - Derivation Rules: Horn subset of FOL (“Datalog”)
    (logical rules of the form $head \leftarrow body$ over literals)
  - Description Logics and OWL (Web Ontology Language)

DATA MANAGEMENT: LOGICAL AND PHYSICAL DATA MODELS

- ER Model, UML Class Diagrams: conceptual models
  There are no existing “ER databases”.
  UML specifications can be implemented in object-oriented databases.
- Relational model:
  - logical model, query languages (relational algebra, SQL)
  - unambiguous mapping from ER to relational model (incl. normalization of tables)
- XML:
  - assumed to be a logical model, but is between a logical and a physical model,
  - many ways how to map an ER model to an XML representation
    (there are many reasonable DTDs for representing the Mondial DB)
    $\Rightarrow$ even though XML data exchange is easy, integration of XML data wrt. different DTDs to an agreed ER model can be troublesome.
  - XML Unicode serialization as a data exchange format
  - the “algebra” underlying XQuery is on a lower level than the relational algebra.
    (structural + content-oriented querying)

COMPARISON

- XML (mapping a domain to a tree structure) is less “semantic” than the relational model (relations for entity types and for relationship types, keys/foreign keys)
- [aside: even the historical network database model (see history part of the SSD/XML lecture) is more semantic than XML]

“Dirty” Points

- XML: subelement relationship has no “name”:
  country/city: subelement means “located in”, while
  city/population: subelement means: has property
- relational model: when n:1 relationships are mapped into a relation (cf. country/capital), the foreign key means “attribute is not a property but a relationship”
  Less dirty, since explicit in the schema.

Network-like models with nodes (objects) and edges (relationships) are more “semantical”

[Aside: “semantic networks” in early approaches to knowledge representation models like KL-ONE etc.]

OUTLOOK: RDF DATA MODEL

- RDF "Resource Description Framework" as a graph-based *logical* data model,
- close to the idea of an "ER database"
  - "things" (graph: nodes) belong to classes/entity types and have an identification (URI - Uniform Resource Identifier)
  - they have attributes (with literal values)
  - there are (binary) relationships (graph: edges). Relationships have names.
- a *node-labeled* and *edge-labeled* data model,
- graphical representation,
- syntactical representations (in plain Unicode, or in an XML(-Unicode) representation of the graph),
- and an SQL-style relational-flavor query language,
- and some other query languages (e.g., again path-based)
  (obvious dialects of OEM/MSL/Lorel and F-Logic apply)

DEVELOPMENT OF LANGUAGES ETC.

Shown in the “history” part of the SSD/XML lecture

XML/XPath/XQuery was influenced by many earlier concepts:

- XML: Network database model, object-oriented database model, earlier self-describing semistructured models (OEM, F-Logic), SGML
- ASCII/Unicode representation/data exchange: ODMG’s OIF (object interchange format)
- XPath: C++ and OQL path expressions (with UNIX notation), F-Logic (conditions embedded into path expressions)
- XQuery: SQL/OQL (clause-based language, but not on a “simple” algebraic foundation)
- not very much theory (except things like tree automata that deal mainly with structural things)
  (recall that there was no theory part in the SSD/XML lecture)

RDF: even more influenced by earlier concepts

- semantic networks: resources (objects and classes), properties and relationships
- relational model/representation: algebra, algebraic query formalisms
- graph model: navigation-based expressions (formally: based on semijoins)
- several Unicode representations
- but: unordered – not suitable for documents.
- in line with *ontology formalisms* for metadata: RDF-RDFS-OWL
- metadata: means logic-based *model theory* with *reasoning*
  (note: constraints etc. in the relational model are also a simple form of model theory)
  - RDF: no reasoning. Data is just data.
    The world is expressed in a single table with only three columns!
  - RDFS: some restricted reasoning
  - OWL Lite/OWL DL/OWL Full:
    ExpTime, but mostly polynomial/up to the border to undecidability/“everything allowed”, possible to express paradoxes

Chapter 3
Databases vs. Knowledge Bases

Overview

- basic notions of first-order logic: syntax, semantics, model theory, tableau – see slides for Database Theory Lecture
- Recall: the Relational Model is a specialization of First-Order Logic:
  - no function symbols. Only constant literals.

Query Answering in DB vs. KB

- A *database (state)* is (e.g.) a *relational structure*.

  Check whether a formula holds there, or for which values of X it holds (which is then a query).

  The *semantics* of a database is the *current database state* (use algebraic evaluation).

- A (first-order) *knowledge base* is a set of closed (first-order) formulas. It contains *facts*, but also other *formulas*.

  We are interested whether a knowledge base $\mathcal{K}$ *implies* a fact or a formula F. This means, if for *all* models $\mathcal{M}$ of $\mathcal{K}$, F must be true in $\mathcal{M}$.

  The semantics of a knowledge base (or in general a set of formulas) is given by a *model theory*:

  Deductive Databases/Datalog: minimal model, stratified model, well-founded model, stable models (with specific algorithms)

  First-Order Logic: all models ("traditional" reasoning)

- an intermediate form occurs when a database is extended by axiomatic formulas (subclasses etc.) or rules that can be used to derive additional facts.
  Then, the semantics is given by the model(s) of the database state and the rules.
  Is the (Semantic) Web more like a database or more like a knowledge base?

LOGICAL MODELING OF ONTOLOGIES

- common "recipe" for mapping ER diagrams to the relational model (cf. MONDIAL):
  - n-ary entity tables with primary keys,
  - n-ary tables for (mostly binary) n:m-relationships with foreign keys,
  - integrate n:1 relationships with the n side,
  - modeling subclass relationships is not a primary concern of the "recipe".
- Alternative (storage) variants, traditionally rather seen as physical data models:
  - column stores,
  - "vertically partitioned": two-column tables with format (id,value) for each attribute or binary relationship; introducing "object" ids.
    - saves space if null values are frequent
    - lots of joins,
    - smaller tuples,
    - less projections
    - better suited for Web context where properties of the same object can be stored at different locations
    - coincides with the modeling in Description Logics

EXAMPLE: AXIOMATIZATION OF THE "COMPANY" ONTOLOGY

Consider again the ER diagram from Slide 22.

- give the *first-order signature* Σ of the ontology,
- formalize the constraints given in the
  - subclass constraints
  - range and domain constraints
  - cardinality constraints

  and
  - additional constraints/definitions that cannot be expressed by the ER model.

  (this set of formulas is called a first-order "theory" or "axiomatization" of the ontology)
- express the instance level as an interpretation of the signature Σ.

Example: Signature

- Classes are represented by unary predicates: Emp/1, Mgr/1, AMgr/1, TMgr/1, Dept/1.
- Attributes are represented by binary predicates: name/2, salary/2 (optionally, this could be modeled by unary functions)
- (binary) relationships are represented by binary relationships: wf/2, mg/2, sub/2.

Thus,
$\Sigma_{company} =$ {Emp/1, Mgr/1, AMgr/1, TMgr/1, Dept/1, name/2, salary/2, wf/2, mg/2, sub/2}.

Example: Subclass Constraints

$$\forall x : \mathsf{Mgr}(x) \rightarrow \mathsf{Emp}(x) \ ,$$
$$\forall x : \mathsf{AMgr}(x) \rightarrow \mathsf{Mgr}(x) \ ,$$
$$\forall x : \mathsf{TMgr}(x) \rightarrow \mathsf{Mgr}(x) \ ,$$
$$\forall x : \mathsf{Mgr}(x) \rightarrow (\mathsf{AMgr}(x) \vee \mathsf{TMgr}(x)) \ \text{ since declared as covering}$$

Example: Domain and Range Constraints

$$\forall x : (\exists n : \mathsf{name}(x, n) \rightarrow (\mathsf{Emp}(x) \vee \mathsf{Dept}(x))) \ ,$$
$$\forall x : (\exists s : \mathsf{salary}(x, s) \rightarrow (\mathsf{Emp}(x))) \ ,$$
$$\forall x : \forall y : (\mathsf{sub}(x, y) \rightarrow (\mathsf{Emp}(x) \wedge \mathsf{Mgr}(y))) \ ,$$
$$\forall x : \forall d : (\mathsf{wf}(x, d) \rightarrow (\mathsf{Emp}(x) \wedge \mathsf{Dept}(d))) \ ,$$
$$\forall x : \forall d : (\mathsf{mg}(y, d) \rightarrow (\mathsf{Mgr}(y) \wedge \mathsf{Dept}(d))) \ .$$

Example: Cardinality Constraints

$$\forall m : (\mathsf{TMgr}(m) \rightarrow \exists d : \mathsf{mg}(m, d)) \ ,$$
$$\forall m, d_1, d_2 : ((\mathsf{mg}(m, d_1) \wedge \mathsf{mg}(m, d_2)) \rightarrow d_1 = d_2) \ ,$$
$$\forall d : (\mathsf{Dept}(d) \rightarrow \exists m : \mathsf{mg}(m, d)) \ ,$$
$$\forall d, m_1, m_2 : ((\mathsf{mg}(m_1, d) \wedge \mathsf{mg}(m_2, d)) \rightarrow m_1 = m_2) \ ,$$
$$\forall d : (\mathsf{Dept}(d) \rightarrow \exists x : \mathsf{wf}(x, d)) \ ,$$
$$\forall x : (\mathsf{Emp}(x) \rightarrow \exists d : \mathsf{wf}(x, d)) \ ,$$
$$\forall x : ((\exists d_1, d_2, d_3, d_4 : \mathsf{wf}(x, d_1) \wedge \mathsf{wf}(x, d_2) \wedge \mathsf{wf}(x, d_3) \wedge \mathsf{wf}(x, d_4)) \rightarrow$$
$$(d_1 = d_2 \vee d_1 = d_3 \vee d_1 = d_4 \vee d_2 = d_3 \vee d_2 = d_4 \vee d_3 = d_4))$$

Example: Further Constraints

- a person is subordinate to the manager of each department he/she works for:

$$\forall x, y, d : \mathsf{wf}(x, d) \wedge \mathsf{mg}(y, d) \wedge x \neq y \rightarrow \mathsf{sub}(x, y)$$

- should we have mg $\subseteq$ wf, or mg $\cap$ wf $= \emptyset$?
  The first is OK: $\forall y, d : \mathsf{mg}(y, d) \rightarrow \mathsf{wf}(y, d)$
- cardinality of "subordinate"? "Every employee has a boss"

$$\forall x : \mathsf{Emp}(x) \rightarrow \exists y : \mathsf{sub}(x, y)$$

  - this causes a semantical problem with the boss: an infinite chain is needed - leading either to only infinite models, or a cycle.
  - add axioms that guarantee transitivity and irreflexivity for "subordinate":
    $\forall x : \neg\mathsf{sub}(x, x)$ and $\forall x, y, z : (\mathsf{sub}(x, y) \wedge \mathsf{sub}(y, z) \rightarrow \mathsf{sub}(x, z))$.
    Then the set of axioms has only one model: Emp is empty, everything is empty.
  - add an axiom that guarantees that the company has at least one employee:
    $\exists x : \mathsf{Emp}(x)$ – then the set of axioms is unsatisfiable.
  - such investigations help to validate an ontology.
    Ontology design tools allow to check for inconsistency, empty classes etc.

Axiomatization of the “company” scenario

Denote the conjunction of the above formulas by $\text{Axioms}_{Company}$.

- For any database/knowledge base $\mathcal{S}$ using this scenario, $\mathcal{S} \models \text{Axioms}_{Company}$ is required.
- a database then describes the individuals and their individual properties in this world.

Example: Instances

- The signature is extended by constant symbols for all *named* elements of the domain:

  $\Sigma_{my\_company} := \Sigma_{company} \cup$ {Alice/f0, Bob/f0, John/f0, Mary/f0, Tom/f0, ..., Sales/f0, ...}

  (Note: the signature symbols are capitalized, wereas alice, bob etc denote the elements of the domain).

- first-order structure $\mathcal{S} = (I, \mathcal{D})$ as ...

- Domain $\mathcal{D}$ = {alice, bob, john, mary, tom, larry, sales, prod, mgm}

- map constant symbols (nullary function symbols) to $\mathcal{D}$:
  I(Alice) = alice, I(Bob) = bob, ..., I(Sales) = sales, ...

- map unary predicates to subsets of the domain $\mathcal{D}$:
  I(Emp) = {alice, bob, john, mary, tom, larry}, I(Mgr) = ..., I(Dept) = ... , ...,

- map binary predicates to subsets of $\mathcal{D} \times \mathcal{D}$:
  I(wf) = {(alice, sales), (mary, sales), (larry, sales), (bob, prod), (bob, sales),
  (tom, prod), (john, mgm)}
  I(mg), I(sub), I(name), I(salary) see Slide 24.

Example: Instances (Cont'd)

The axiomatization of "my company" with the given individuals is then given as the conjunction of

- all above constraints (general axiomatization of a company) and
- literal formulas describing the individuals:

$\text{Axioms}_{\mathfrak{C}_{Company}} \ldots \wedge \text{Emp}(\text{Alice}) \wedge \text{Emp}(\text{Bob}) \wedge \ldots \wedge \text{Dept}(\text{Sales}) \wedge \ldots \wedge \text{Mgr}(\text{Alice}) \wedge \ldots \wedge \text{wf}(\text{Alice}, \text{Sales}) \wedge \ldots \wedge \text{mg}(\text{Alice}, \text{Sales}) \wedge \ldots \wedge \text{sub}(\text{Mary}, \text{Alice}) \wedge \ldots.$

Example: Instances (Alternative)

- alternatively, instead of a signature extensions, all individuals can be described by *existentially* quantified variables:

$\text{Axioms}_{\mathfrak{C}_{Company}} \wedge$
$\exists a, b, \ldots, s, \ldots : (\text{name}(a, \text{“Alice”}) \wedge \ldots \wedge \text{Emp}(a) \wedge \text{Emp}(b) \wedge \ldots \wedge \text{Dept}(s) \wedge \ldots \wedge \text{Mgr}(a) \wedge \ldots \wedge \text{wf}(a, s) \wedge \ldots \wedge \text{mg}(a, s) \wedge \ldots \wedge \text{sub}(m, a) \wedge \ldots)$

3.1 DB vs. KB: Closed World vs. Open World

Consider the following formula F:

$$\begin{aligned} F \quad \equiv \quad & person(\text{“John”}, 35) \wedge person(\text{“Alice”}, 10) \wedge person(\text{“Bob”}, 8) \wedge \\ & \wedge person(\text{“Carol”}, 12) \wedge person(\text{“Jack”}, 65) \wedge \\ & \wedge child(\text{“John”}, \text{“Alice”}) \wedge child(\text{“John”}, \text{“Bob”}) \wedge \\ & \forall X, Y : (\exists Z : (child(Z, X) \wedge child(Z, Y) \wedge X \neq Y) \rightarrow sibling(X, Y)) \end{aligned}$$

- Does $child(\text{“John”}, \text{“Bob”})$ hold? – obviously yes.
- Does $G{:}\equiv sibling(\text{“Alice”}, \text{“Bob”})$ hold?
  - (Relational) Database: $sibling$ is a view. The answer is “yes”.
  - FOL KB: for all models $\mathcal{M}$ of F, G holds. Thus, $F \models sibling(\text{“Alice”}, \text{“Bob”})$.
- What about $G{:}\equiv sibling(\text{“Alice”}, \text{“Carol”})$?
  - (Relational) Database: no. For the database state $\mathbf{D}$, $\mathbf{D} \models \neg sibling(\text{“Alice”}, \text{“Carol”})$.
  - FOL KB: there is a model $\mathcal{M}_1$ of F, where $\mathcal{M}_1 \not\models G$, but there is also a model $\mathcal{M}_2$ of F, where $\mathcal{M}_2 \models G$ (e.g., add the tuple $(\text{“John”}, \text{“Carol”})$ to the interpretation of $child$).

  For the Web, $child(\text{“John”}, \text{“Carol”})$ can e.g. be contributed by another Web Source.

DB vs. KB: Closed World vs. Open World

- What about $G :\equiv child(\text{“John”}, \text{“Jack”})$?
  - (Relational) Database: no. For the database state $\mathbf{D}$, $\mathbf{D} \not\models child(\text{“John”}, \text{“Jack”})$.
  - FOL KB: there is a model $\mathcal{M}_1$ of F, where $\mathcal{M}_1 \not\models G$, but there is also a model $\mathcal{M}_2$ of F, where $\mathcal{M}_2 \models G$.
- Obviously, the KB does not know that a child cannot be older than its parents.

  Add a constraint to F, obtaining F':

  $$F' :\equiv F \wedge \forall P, C, A_1, A_2, : (person(P, A_1) \wedge person(C, A_2) \wedge child(P, C)) \rightarrow A_1 > A_2$$

  - database: this assertion would prevent to add $child(\text{“John”}, \text{“Jack”})$ to the database.
  - for the KB, $F' \models \neg child(\text{“John”}, \text{“Jack”})$ allows to *infer* that Jack is not the child of John.

  Such information can be given with the *ontology* of a domain.

DB vs. KB: Closed World vs. Open World

- the Database Model Theory is called *"Closed World"*: things that are not known to hold are *assumed* not to hold.
- the FOL semantics is called *"Open World"*: things that are not known to be true or false are considered to be *possible*.

Consequences on Negation

- in databases there is no explicit negation. It is not necessary to specify that Jack is *not* a child of John.
- in a KB, it would be necessary to state ... $\wedge \neg child(\text{"John"}, X)$ for all persons who are known not to be children of John.
  Additional constraints: extend the ontology, e.g., by stating that a person has exactly two parents – then all others cannot be parents – works only for persons whose parents are known. Similarly for the "age" constraint from the previous slide.
- note that the semantics of universal quantification ($\forall$) is also effected: $\forall X : \phi$ is equivalent to $\neg\exists X : \neg\phi$.

REASONING IN PRESENCE OF NEGATION

Obtaining new information (e.g., by finding another Web Source) has different effects on Open vs. Closed world:

- Closed world: conclusions drawn before – "Carol is not a child of John", or "John has exactly two children" from less information can become invalid.

  This kind of reasoning is *nonmonotonic*

- Open world: everything which is not known explicitly is taken into account to be possible (by considering all possible models).

  This kind of reasoning is *monotonic*:

$$\text{Knowledge}_1 \subseteq \text{Knowledge}_2 \Rightarrow \text{Conclusions}_1 \subseteq \text{Conclusions}_2$$

- Open World can be combined with other forms of nonmonotonic reasoning, e.g., Defaults: "usually, birds can fly". Knowing that Tweety is a bird allows to conclude that it flies.

  Obtaining the information that Tweety is a penguin (which can usually not fly) leads to invalidation of this conclusion.

The current Semantic Web research mainstream prefers Open World without default reasoning.

COMPARISON, MOTIVATION ETC.

Database vs. FOL

Relational Databases	relational schema	tuples	SQL queries	closed world
FOL	signature (predicates +functions)	facts (atoms)	$\mathcal{S} \models \phi$? (yes/no or answer $\psi \models \phi$? variable bindings)	mostly: open world sometimes closed world

Situations and tasks

Given	what to do	how?
facts/database	does $p(\ldots)$ hold in the DB? SQL query	by combining data
facts+constraints (SQL assertions or FOL formulas)	additionally: test if constraints satisfied	equivalent to first situation (query for violating tuples)
facts (DB) rules (KB)	does $p(\ldots)$ hold in DB+rules?	DB+views application of rules
facts (DB) knowledge base KB as FOL formulas	is a formula ϕ entailed by DB+KB?	reasoning, entailment, KB $\models \phi$?

VALIDITY AND DECIDABILITY

- preferably use a decidable logic/formalism
- with a *complete* calculus/reasoning mechanism
- Propositional logic: decidable
- First-order logic: undecidable
- Horn subset of FOL
  (logical rules of the form $head \leftarrow body$ over literals); decidable
- 2-variable-subset of FOL: decidable
- Description Logic subsets of FOL: range from decidable to undecidable

3.2 Reasoning about Ontologies

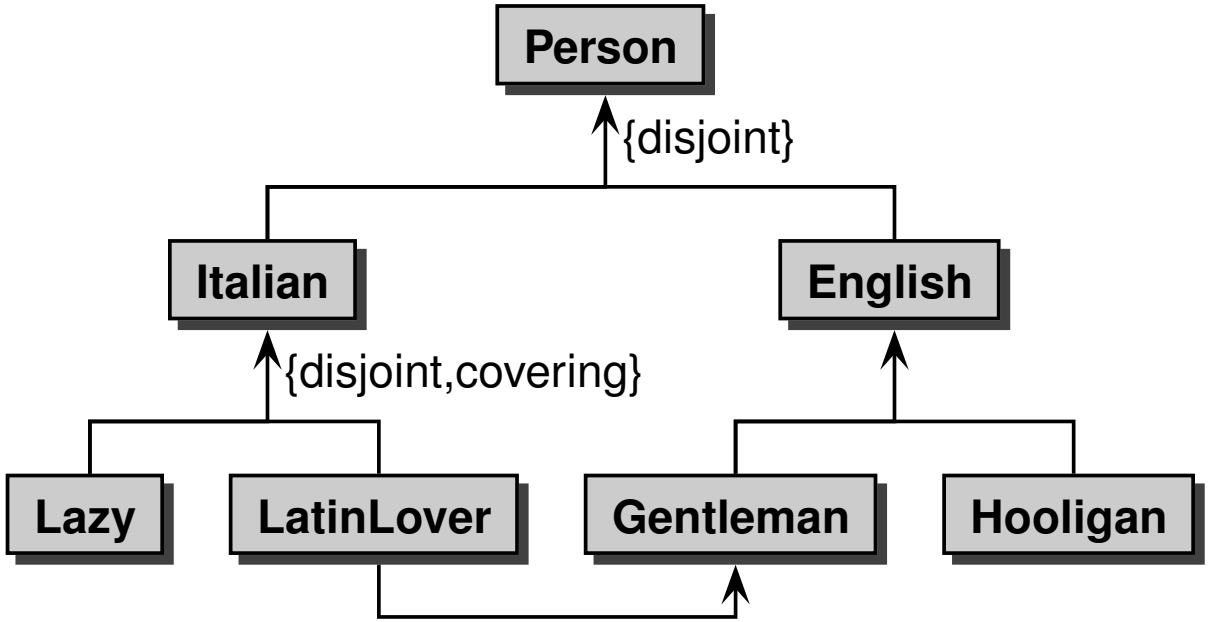
Traditional Description Logics Research

Given an ontology,

- a class is *inconsistent* if it denotes the empty set in every model,
- a class C is a *subclass* of D if the extension of C is a subset of the extension of D in every model,
- two classes are *equivalent* if they denote the same set in every model,
- a constraint is *entailed* by an ontology if it holds in every model.

$\Rightarrow$ focus on metadata ("T-Box"); query answering wrt. instances ("A-Box") did not play a big role.

EXAMPLE: ITALIANS AND ENGLISHMEN



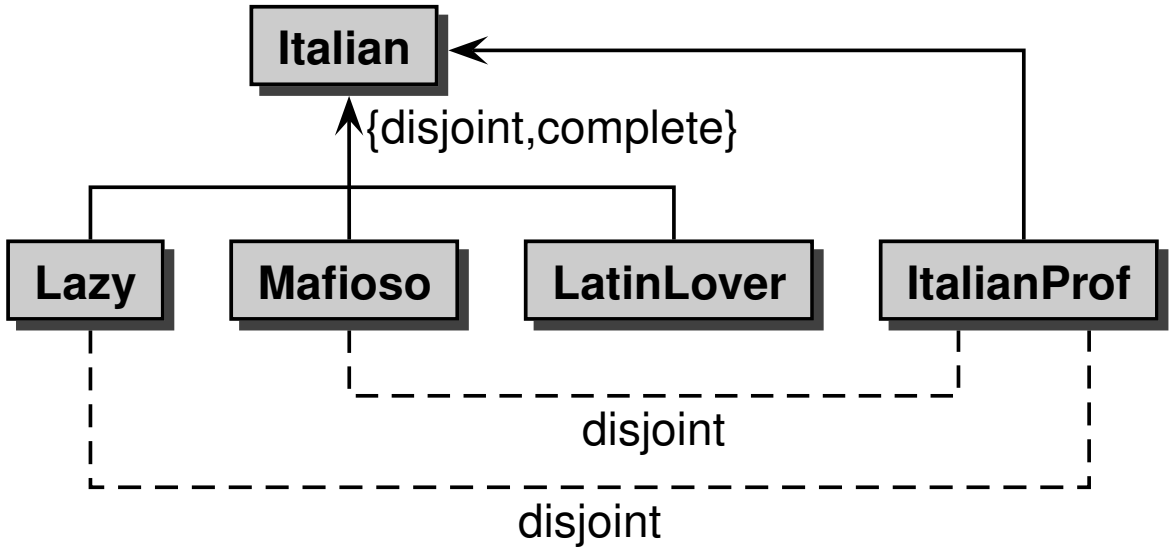
Exercise: write down as concise as possible *everything* that is implied by this ontology in text, set theory and first-order logic.

[by Enrico Franconi, REWERSE Summer School 2005]

[see Slide 71 for an excerpt and a relevant proof]

.

EXAMPLE: ITALIAN PROFESSORS

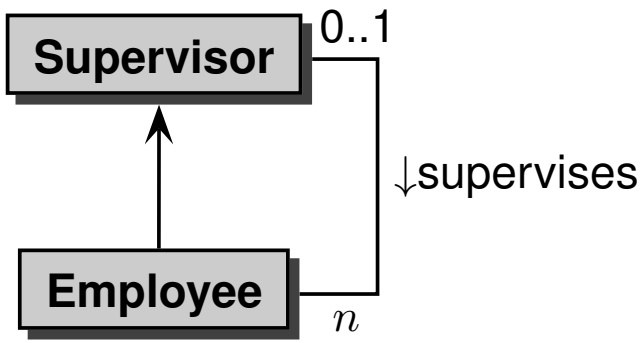


Exercise: write down as concise as possible *everything* that is implied by this ontology in text, set theory and first-order logic.

[by Enrico Franconi, REWERSE Summer School 2005]

EXAMPLE: THE DEMOCRATIC COMPANY

Every employee is also a supervisor. Every supervisor supervises 2 employees.

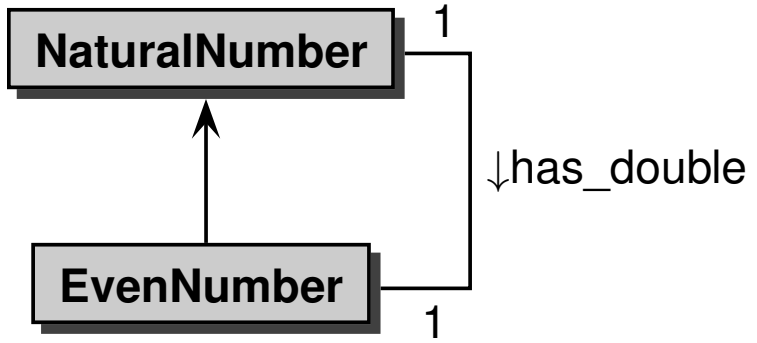


Exercise: Consider $n = 1$ and $n = 2$. Write down as concise as possible *everything* that is implied by this ontology in text, set theory and first-order logic.

[by Enrico Franconi, REWERSE Summer School 2005]

EXAMPLE: BIJECTION - HOW MANY NUMBERS

Every natural number is related to its double which is an even number.



- the classes “NaturalNumber” and “EvenNumber” contain the same number of elements (standard model: infinitely many),
- the same applied to a *finite* domain implies that NaturalNumber $\equiv$ EvenNumber (which e.g. holds in the *cyclic* modulo rings Z_3, Z_5 etc.).
  (note: in these rings, every element is even!)

[by Enrico Franconi, REWERSE Summer School 2005]

3.3 Reasoning

- show $F \models G$: prove that $F \wedge \neg G$ is unsatisfiable:
  try to construct an example ("witness") for $F \wedge \neg G$ by a systematic algorithm.
  (below: e.g. using a tableau calculus)
- show satisfiability of a knowledge base/ontology:
  try to construct an example.
- answer a query (= a conjunctive formula Q with free variables) against a knowledge base given as a formula F:
  Construct counterexamples for $F \wedge \neg Q$; each of them is an answer.

$\Rightarrow$ one algorithm is sufficient.

RECALL: FIRST ORDER TABLEAU CALCULUS

- Systematic construction of an interpretation of a formula.
- Goal: show that this is not possible. Otherwise a counterexample is generated.
- counterexamples can be interpreted as answers to a query.

Start the tableau with a set $\mathcal{F}$ of formulas:

$$\frac{\text{input set } \mathcal{F}}{F \quad \text{for all } F \in \mathcal{F}}$$

The tableau is then extended by expansion rules.

Tableau Rules

α-rule (conjunctive): $\dfrac{F \wedge G}{\begin{array}{c} F \\ G \end{array}}$ $\qquad$ $\dfrac{\neg(F \vee G)}{\begin{array}{c} \neg F \\ \neg G \end{array}}$

β-rule (disjunctive): $\dfrac{F \vee G}{F \mid G}$ $\qquad$ $\dfrac{\neg(F \wedge G)}{\neg F \mid \neg G}$

γ-rule (universal): $\dfrac{\forall x : F}{F[X/x]}$ $\qquad$ $\dfrac{\neg\exists x : F}{\neg F[X/x]}$

where X is a new variable.

δ-rule (existential): $\dfrac{\exists x : F}{F[f(\mathsf{free}(T))/x]}$ $\qquad$ $\dfrac{\neg\forall x : F}{\neg F[f(\mathsf{free}(T))/x]}$

where f is a new *Skolem function symbol* (after the Norwegian logician Thoralf Skolem) and T is the current branch of the tableau.

Closure Rule:

$$\frac{\begin{array}{c} \sigma(A) \\ \neg\sigma(A) \end{array}}{\bot}$$

apply σ to the whole tableau.

Result

Definition 3.1

A branch T in a tableau $\mathcal{T}$ is *closed*, if it contains the formula $\bot$.

A tableau $\mathcal{T}$ is *closed* if every branch is closed. □

Correctness

Definition 3.2

A tableau $\mathcal{T}$ is *satisfiable* it there exists an interpretation $\mathcal{S} = (U, I)$ such that for every assignment of the free variables there is a branch T in $\mathcal{T}$ such that $(\mathcal{S}, \beta) \models T$ holds. □

Theorem 3.1

If a tableau $\mathcal{T}$ is satisfiable, and $\mathcal{T}'$ is obtained from $\mathcal{T}$ by application of one of the above rules, then $\mathcal{T}'$ is also satisfiable. □

TABLEAU CALCULUS: EXAMPLE/EXERCISE

Consider $\text{Axioms}_{\mathcal{C}ompany} \wedge \text{mg}(\text{Alice}, \text{Sales})$. (Slides 38 ff.).

Does

$(\forall y, d : (\text{mg}(y, d) \rightarrow \text{wf}(y, d))) \wedge (\forall x : \forall d : (\text{wf}(x, d) \rightarrow (\text{Emp}(x) \wedge \text{Dept}(d)))) \wedge \text{mg}(\text{Alice}, \text{Sales})$

imply $\text{Emp}(\text{Alice})$?

see next slide ...

Tableau Calculus: Example

$\forall y, d : (\mathsf{mg}(y, d) \rightarrow \mathsf{wf}(y, d))$

$\forall x : \forall d : (\mathsf{wf}(x, d) \rightarrow (\mathsf{Emp}(x) \wedge \mathsf{Dept}(d)))$

$\mathsf{mg}(\mathsf{Alice}, \mathsf{Sales})$

$\neg\mathsf{Emp}(\mathsf{Alice})$ $(**)$

|

$\mathsf{mg}(Y_1, D_1) \rightarrow \mathsf{wf}(Y_1, D_1)$

/ \

$\neg\mathsf{mg}(Y_1, D_1)$ | $\mathsf{wf}(Y_1, D_1)$

$\square\{Y_1 \rightarrow \mathsf{Alice}, D_1 \rightarrow \mathsf{Sales}\}$ | $\mathsf{wf}(\mathsf{Alice}, \mathsf{Sales})$

|

$\mathsf{wf}(X_2, D_2) \rightarrow (\mathsf{Emp}(X_2) \wedge \mathsf{Dept}(D_2))$

/ \

$\neg\mathsf{wf}(X_2, D_2)$ | $\mathsf{Emp}(X_2) \wedge \mathsf{Dept}(D_2)$

$\square\{X_2 \rightarrow \mathsf{Alice}, D_2 \rightarrow \mathsf{Sales}\}$ | $\mathsf{Emp}(\mathsf{Alice}) \wedge \mathsf{Dept}(\mathsf{Sales})$

$\mathsf{Emp}(\mathsf{Alice})$ $(**)$

$\mathsf{Dept}(\mathsf{Sales})$

$\square(**)$

EXAMPLE: TABLEAU EXPANSION FOR AN EXISTENTIAL VARIABLE

Consider again the Company scenario. Show: for every employee x, there is an employee y ($x = y$ allowed) such that sub(x, y) holds. (sketch: for every employee x there is a at least a "primary" department $f_{dept}(x)$ where this person works, and every department d has a manager $f_{mg}(d)$ that manages the department and that thus is a subordinate of x.

Note that in case that x works in several departments, any of them can be chosen for $f_{dept}(x)$. e is subordinate to $f_{mg}(f_{dept}(x))$.

Tableau: next slide.

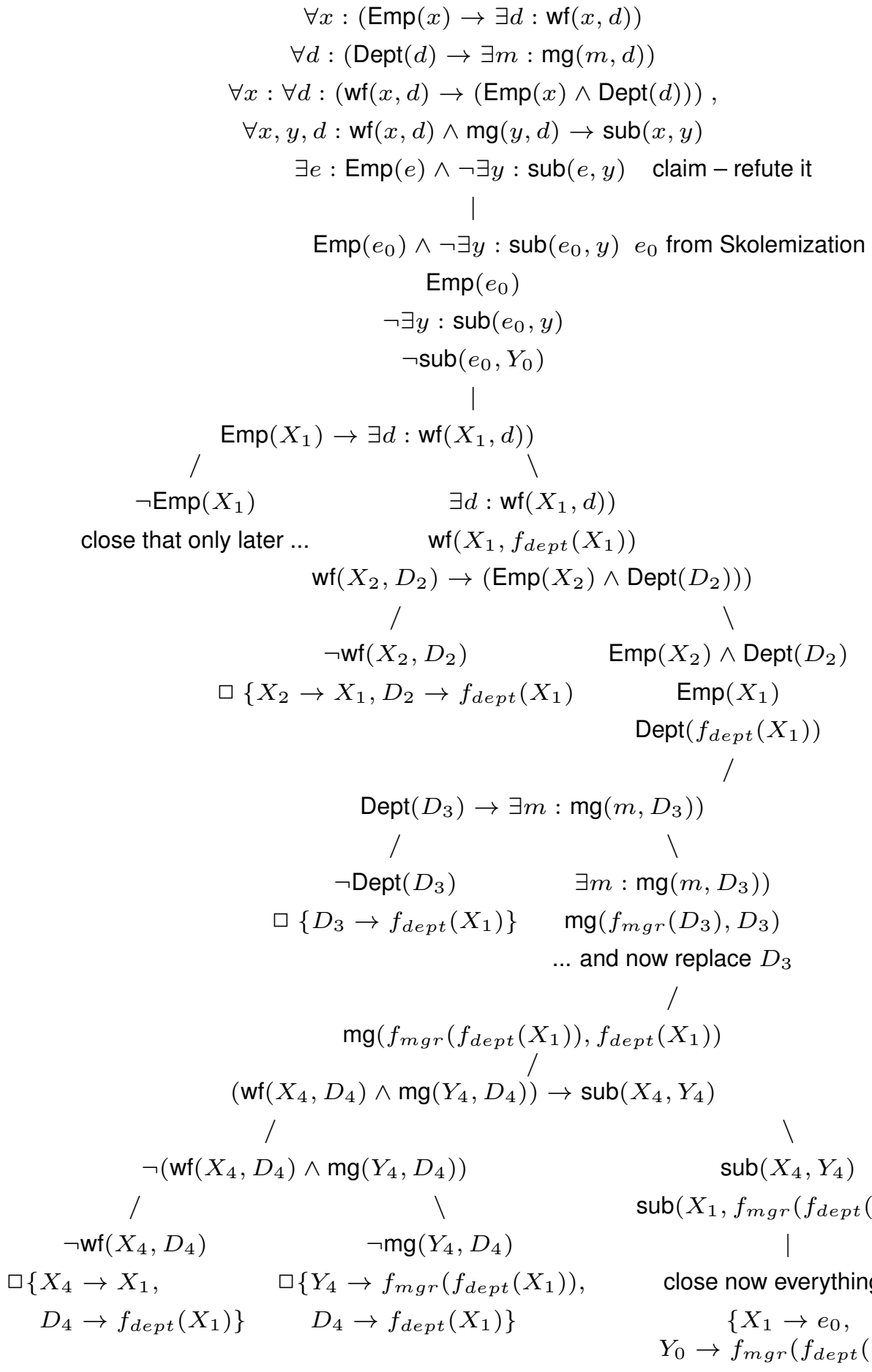

$\forall x : (\mathsf{Emp}(x) \rightarrow \exists d : \mathsf{wf}(x, d))$
$\forall d : (\mathsf{Dept}(d) \rightarrow \exists m : \mathsf{mg}(m, d))$
$\forall x : \forall d : (\mathsf{wf}(x, d) \rightarrow (\mathsf{Emp}(x) \wedge \mathsf{Dept}(d)))$,
$\forall x, y, d : \mathsf{wf}(x, d) \wedge \mathsf{mg}(y, d) \rightarrow \mathsf{sub}(x, y)$
$\exists e : \mathsf{Emp}(e) \wedge \neg \exists y : \mathsf{sub}(e, y)$ claim – refute it
$\mathsf{Emp}(e_0) \wedge \neg \exists y : \mathsf{sub}(e_0, y)$ e_0 from Skolemization
$\mathsf{Emp}(e_0)$
$\neg \exists y : \mathsf{sub}(e_0, y)$
$\neg \mathsf{sub}(e_0, Y_0)$
$\mathsf{Emp}(X_1) \rightarrow \exists d : \mathsf{wf}(X_1, d))$
$\neg \mathsf{Emp}(X_1)$
close that only later ...
$\exists d : \mathsf{wf}(X_1, d))$
$\mathsf{wf}(X_1, f_{dept}(X_1))$
$\mathsf{wf}(X_2, D_2) \rightarrow (\mathsf{Emp}(X_2) \wedge \mathsf{Dept}(D_2)))$
$\neg \mathsf{wf}(X_2, D_2)$
$\square \{X_2 \rightarrow X_1, D_2 \rightarrow f_{dept}(X_1)$
$\mathsf{Emp}(X_2) \wedge \mathsf{Dept}(D_2)$
$\mathsf{Emp}(X_1)$
$\mathsf{Dept}(f_{dept}(X_1))$
$\mathsf{Dept}(D_3) \rightarrow \exists m : \mathsf{mg}(m, D_3))$
$\neg \mathsf{Dept}(D_3)$
$\square \{D_3 \rightarrow f_{dept}(X_1)\}$
$\exists m : \mathsf{mg}(m, D_3))$
$\mathsf{mg}(f_{mgr}(D_3), D_3)$
... and now replace D_3
$\mathsf{mg}(f_{mgr}(f_{dept}(X_1)), f_{dept}(X_1))$
$(\mathsf{wf}(X_4, D_4) \wedge \mathsf{mg}(Y_4, D_4)) \rightarrow \mathsf{sub}(X_4, Y_4)$
$\neg(\mathsf{wf}(X_4, D_4) \wedge \mathsf{mg}(Y_4, D_4))$
$\mathsf{sub}(X_4, Y_4)$
$\mathsf{sub}(X_1, f_{mgr}(f_{dept}(X_1)))$
$\neg \mathsf{wf}(X_4, D_4)$
$\square \{X_4 \rightarrow X_1,$
$D_4 \rightarrow f_{dept}(X_1)\}$
$\neg \mathsf{mg}(Y_4, D_4)$
$\square \{Y_4 \rightarrow f_{mgr}(f_{dept}(X_1)),$
$D_4 \rightarrow f_{dept}(X_1)\}$
close now everything by
$\{X_1 \rightarrow e_0,$
$Y_0 \rightarrow f_{mgr}(f_{dept}(e_0))\}$

TABLEAU CALCULI: APPLICATION FOR (DATABASE) QUERY ANSWERING

Consider two binary predicates p and q (in a deductive DB context called "EDB" (extensional database)) and a derived unary predicate r as $\forall x : (r(x) \leftrightarrow \exists y : (p(x,y) \vee q(x,y)))$, and the query $? - r(Z)$.

$\forall x : (r(x) \leftrightarrow \exists y : (p(x,y) \vee q(x,y)))$

$\neg r(Z)$ add the negated query with a free variable

$\forall x : (r(x) \leftarrow \exists y : (p(x,y) \vee q(x,y)))$ need only "$\leftarrow$" direction to close

$\forall x : (r(x) \rightarrow \exists y : (p(x,y) \vee q(x,y)))$ not needed

$r(X) \leftarrow \exists y : (p(X,y) \vee q(X,y))$

$r(X)$
$\square\{Z \rightarrow X\}$

$\neg\exists y : (p(X,y) \vee q(X,y))$

$\neg p(X,Y)$ can be closed with any tuple $p(x,y)$

$\neg q(X,Y)$ can be closed with any tuple $q(x,y)$

- collect all substitutions that can be used to close the tableau:
  $\rho[X \rightarrow Z](\pi[\$1](p) \cup \pi[\$1](q))$ (where $\$n$ denotes the nth column)

Tableau Calculi for (Database) Query Answering (same example, other way)

Same as before, only that (the same r) is now expressed in Datalog style by two rules:

- different ways how to close the tableau $\Rightarrow$ union of answer substitutions

$$\forall x, y : (p(x, y) \rightarrow r(x))$$
$$\forall x, y : (q(x, y) \rightarrow r(x))$$
$$\neg r(Z) \quad \text{add the negated query with a free variable}$$
$$p(X_1, Y_1) \rightarrow r(X_1)$$
$$q(X_2, Y_2) \rightarrow r(X_2)$$

two possibilities which rule to expand:

the first one with p ...

$\neg p(X_1, Y_1)$ can be closed with any tuple $p(x, y)$

$r(X_1)$ $\square\{Z \rightarrow X_1\}$

... or the second one with q

$\neg q(X_2, Y_2)$ can be closed with any tuple $q(x, y)$

$r(X_2)$ $\square\{Z \rightarrow X_2\}$

- again, collect all substitutions that can be used to close the tableau:
  $\rho[X \rightarrow Z](\pi[\$1](p)) \cup \rho[X \rightarrow Z](\pi[\$1](q))$ (again, $\$n$ denotes the nth column)

Example: Existential Knowledge

- the answer substitution can comprise the application of a Skolem function. Then, the "answer" can only be described as a thing that satisfies a certain existential formula.

Example: Tableau with Existential Knowledge

$\forall x : (\text{person}(x) \rightarrow (\exists y : \text{person}(y) \wedge \text{father}(x, y)))$

$\forall x, y, z : ((\text{father}(x, y) \wedge \text{father}(y, z)) \rightarrow \text{grandfather}(x, z))$

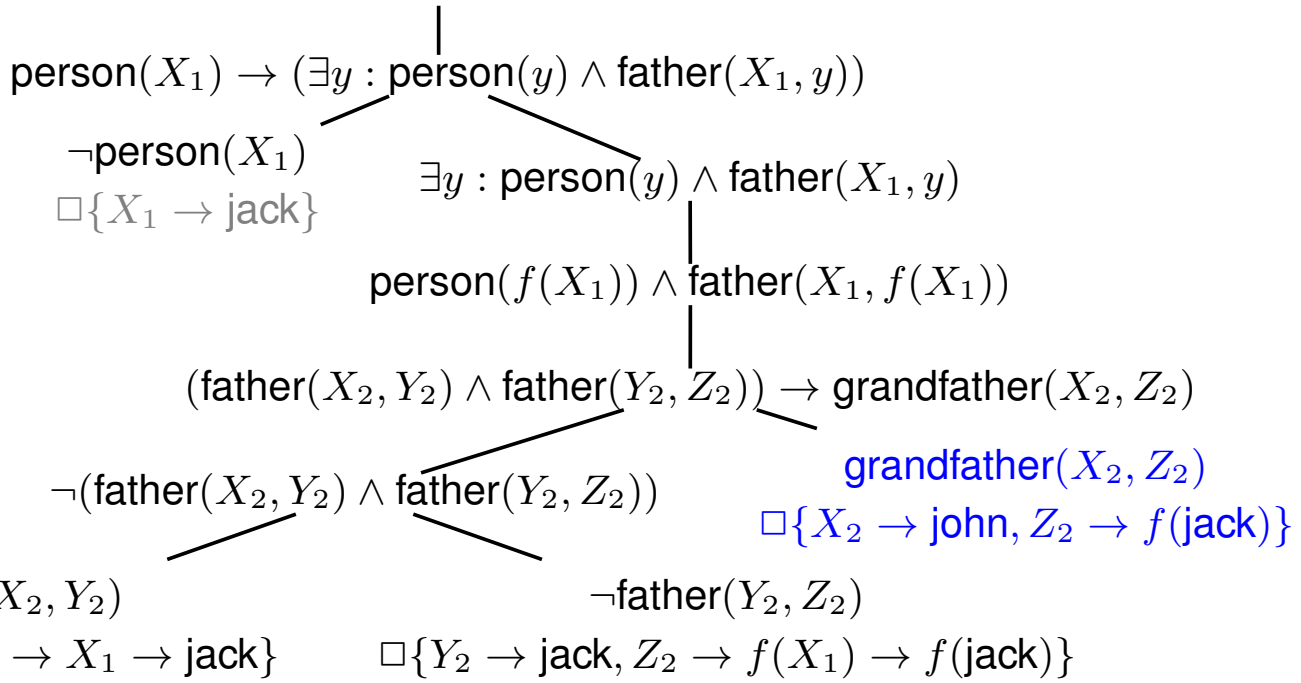
person(john)

person(jack)

father(john, jack)

Negated query: $\neg$ grandfather(john, X)

Human-reasoning-inspired proof: Jack's father is John's grandfather



close all with

$\{X_1 \rightarrow \text{jack},\ X_2 \rightarrow \text{john},\ Y_2 \rightarrow X_1 \rightarrow \text{jack},\ Z_2 \rightarrow f(X_1) \rightarrow f(\text{jack}),\ X \rightarrow Z_2 \rightarrow f(X_1) \rightarrow f(\text{jack})\}$.

Example: Existential Knowledge

- the answer contains an instance of the introduced skolem function $f(\_)$ (which stands for "father of _") to describe a person for which no constant identifier is known.
- Are there further answers? - yes:
  - instantiate the rule
    $\forall x : (\text{person}(x) \rightarrow (\exists y : \text{person}(y) \wedge \text{father}(x, y)))$
    for John, get a new instance of person $f(\text{john})$, instantiate the rule again for $f(\text{john})$ and get $f(f(\text{john}))$ and grandfather(john, $f(f(\text{john})))$.
  - here, functionality of father is required:
    $\forall x, y_1, y_2 : \text{father}(x, y_1) \wedge \text{father}(x, y_2) \rightarrow y_1 = y_2$.
  - equality must then be used to find a most simple expression for the answers.
- more general case: query ?- grandfather(X, Y) for all answers:
  - instantiations of the "father-generating" rule lead to infinitely many father-chains for every known instance of person (using the skolem function)
  - infinitely many answers.

$\Rightarrow$ tableau expansion for only existentially known instances must be blocked when nothing "relevant" is known about them.

$\Rightarrow$ Preview: SPARQL will *not* return any answers that contain skolem expressions.

TABLEAU CALCULI

- intuitive idea
- can be designed in this way for any logic (modal logics, description logics etc.)
- implementations use more efficient heuristics

EXAMPLES + EXERCISES

- Consider again the italian-vs-english ontology from Slide 52. Consider the statement “all Italians are lazy”. Prove it or give a counterexample.
- Consider again the italian-professors ontology from Slide 53. Is there anything interesting to prove?

[have also a look at the i•com tool at `http://www.inf.unibz.it/~franconi/icom` which uses a (hidden) Description Logic prover]

Tableau Proof (Example)

Tableau for the italian-vs-english ontology from Slide 52 and the statement “all Italians are lazy”.

$$\forall x : \text{italian}(x) \rightarrow \neg\text{english}(x)\ [1]$$
$$\forall x : \text{english}(x) \rightarrow \neg\text{italian}(x)\ [2]$$
$$\forall x : \text{italian}(x) \rightarrow (\text{lazy}(x) \vee \text{latinlover}(x))\ [3]$$
$$\forall x : \text{lazy}(x) \rightarrow \neg\text{latinlover}(x))\ [4]$$
$$\forall x : \text{latinlover}(x) \rightarrow \neg\text{lazy}(x)\ [5]$$
$$\forall x : \text{latinlover}(x) \rightarrow \text{gentleman}(x)\ [6]$$
$$\forall x : \text{gentleman}(x) \rightarrow \text{english}(x)\ [7]$$
$$\exists x : \text{italian}(x) \wedge \neg\text{lazy}(x)\ \ [8]\ \ \text{(negation of the claim)}$$
$$\mid\ \text{(skolemization of [8])}$$
$$\text{italian}(c) \wedge \neg\text{lazy}(c)$$
$$\text{italian}(c)$$
$$\neg\text{lazy}(c)$$
$$\mid\ \text{(use [3])}$$
$$\forall x : \text{italian}(x) \rightarrow (\text{lazy}(x) \vee \text{latinlover}(x))$$
$$\text{italian}(X_1) \rightarrow (\text{lazy}(X_1) \vee \text{latinlover}(X_1))$$
$$/ \qquad\qquad \backslash$$
$$\neg\text{italian}(X_1) \qquad \text{lazy}(X_1) \vee \text{latinlover}(X_1)$$
$$\square\ \{X_1 \rightarrow c\} \qquad \text{lazy}(c) \vee \text{latinlover}(c)$$
$$/ \qquad\quad \backslash$$
$$\text{lazy}(c) \qquad \text{latinlover}(c)$$
$$\square$$

Continue right branch using [6], [7] and finally [1] or [2].

First-Order Logic Decision Procedures

- calculi (=algorithms) for checking if $F \models G$
  (often by proving that $F \wedge \neg G$ is unsatisfiable)
- write $F \vdash_C G$ if calculus C proves that $F \models G$.
- Correctness of a calculus: $F \vdash_C G \Rightarrow F \models G$
- Completeness of a calculus: $F \models G \Rightarrow F \vdash_C G$
- there are complete calculi and proof procedures for propositional logic (e.g., Tableau Calculus or Model Checking)
- if a logic is undecidable (like first-order logic) then there cannot be any complete calculus!

What to do?

$\Rightarrow$ use a decidable logic (i.e., weaker than FOL).

$\Rightarrow$ use an undecidable logic and a correct, but incomplete calculus.

INFERENCE SYSTEMS

- use *inference rules* (dt.: Schlussregeln) as patterns
- "Modus Ponens": $$\frac{\textit{head} \leftarrow \textit{body} \quad , \quad \textit{fml} \quad , \quad \sigma(\textit{fml}) \rightarrow \sigma(\textit{body})}{\sigma(\textit{head})}$$
- simple case: Datalog-style rules (many other systems use similar derivation rules)

$$\frac{\textit{head\_atom} \leftarrow \textit{atom}_1, \ldots, \textit{atom}_n \ , \text{ ground } \textit{atom'}_1, \ldots, \textit{atom'}_n \ , \text{ for all } i\text{: } \sigma(\textit{atom}_i) = \textit{atom'}_i}{\sigma(\textit{head\_atom})}$$

- Resolution Calculus:
  a *clause* is a set of literals. Clause resolution takes two clauses that contain contradictory literals:

$$\frac{\ell_1 \vee \ldots \vee \boxed{\ell_i} \vee \ldots \vee \ell_k \quad , \quad \ell_{k+1} \vee \ldots \vee \boxed{\neg \ell_{k+j}} \vee \ldots \vee \ell_{k+m} \quad , \quad \boxed{\sigma(\ell_i) = \sigma(\ell_{k+j})}}{\sigma(\ell_1 \vee \ldots \vee \ell_{i-1} \vee \ell_{i+1} \vee \ldots \vee \ell_k \vee \ell_{k+1} \vee \ldots \vee \ell_{k+j-1} \vee \ell_{k+j+1} \vee \ldots \vee \ell_{k+m})}$$

- simple case: unit resolution: $j = m = 1$
- since a derivation rule *head*$\leftarrow$*body* is equivalent to $\neg$*body* $\vee$ *head*, the bottom-up evaluation of derivation rules is a special case of resolution.

SPECIALIZED INFERENCE RULES

- Modus ponens and resolution are purely syntactical, general-purpose rules that do not depend on the semantics of the literals.
- Logics can extend them with specialized semantical rules that apply to special "predicates".
  - class hierarchy: classes and subclasses, transitivity
  - signatures
  - cardinalities

  Examples: Default Logic (Reiter, 1980; only built-in reasoning), Description Logics (late 1980s; only built-in reasoning), F-Logic (Kifer, Lausen, 1989; Datalog style + built-in), Hybrid Logics (combining Description Logics with derivation)
- the latter *built-in axioms* for certain fixed notions are referred to as "the model theory of a certain logic".

SUMMARY

Above: short introduction to some Knowledge Representation formalisms:

- Derivation Rules
- Automated Reasoning

Semantic Web Reasoning

- use mechanisms of logics and “Artificial Intelligence” invented in the 60s-90s
- disappointment about AI in the 90s:
  - promised too much (expert systems etc.)
  - often worked only for toy examples
- further investigations in the 90s
  - better understanding of decidable and tractable fragments (“tractable” = polynomial complexity) and efficiency issues
- design Semantic Web technology according to these investigations.

LEVELS OF INFERENCE

In the following, three levels of inference and knowledge modeling are combined:

1. The underlying inference system:
   choice of certain underlying *expressive logics* with their built-in model theory
   - First-order logic: quite expressive, generic model theory
   - Description logics:
     - less expressive (only unary and binary predicates, quite restricted construction of formulas)
     - some built-in notions + model theory
     - user can exploit these notions
2. a domain ontology (= formulas in the underlying logic that express global properties and constraints of the domain)
   [mainly describing the classes and properties; optionally some “important” individuals]
3. facts that describe a certain state
   [the “Web contents”, talking about individuals; incomplete knowledge]

DATA FORMAT AND REASONING FOR THE SEMANTIC WEB

- data model: intuitive modeling capabilities on the conceptual level
- describe data and metadata
  $\Rightarrow$ metadata notions are also objects of the domain of discourse.
- tailored to the Web: multiple sources describe the same things, semantic interoperability
- data format/representation: syntactic interoperability/data exchange in the Web required.
  $\Rightarrow$ Unicode, preferably an XML representation (then, no special parser is needed).
  (cf. Unicode serialization/representation of the XML tree model)
  for the new data model, an XML-tree representation will be one possible representation of the data model.
- reasoning: decidable fragment of FOL vs. undecidable FOL vs. even more expressiveness (reasoning about metadata)

ASIDE: WHY "FIRST-ORDER"-LOGIC?

Recall:

- there is a domain $\mathcal{D}$. Functions and precidates talk *about* elements of $\mathcal{D}$.
- there is no way to talk *about* functions or predicates.

Higher-Order-Logics

- the elements of the domain $\mathcal{D}$ are "first-order things"
- sets, functions and predicates are "second-order things"
- predicates about predicates are higher-order things
- higher-order logics can be used for reasoning *about* metadata

Example

- Transitivity as a property of predicates is second order:
  $\forall p : \text{transitive}(p) \rightarrow (\forall x, z : (\exists y : (p(x, y) \wedge p(y, z)) \rightarrow p(x, z)))$
  Note that transitivity of *a certain* predicate is first-order:
  $\forall x, z : ((\exists y : (\text{ancestor}(x, y) \wedge \text{ancestor}(y, z))) \rightarrow \text{ancestor}(x, z))$

Aside: Induction Axiom as Example for Second Order Logic

- a well-founded domain d (i.e., a finite set of minimal elements (for which min(d,x) holds) from which the domain can be enumerated by a successor predicate
  (Natural numbers: 1, succ(i,i+1))
- well-founded: unary 2nd-order predicate over sets

$$\forall p, d: \quad (\text{well-founded}(d) \wedge (\forall x : \text{min}(d, x) \rightarrow p(x)) \wedge (\forall x, y : p(x) \wedge \text{succ}(x, y) \rightarrow p(y))) \rightarrow (\forall x : d(x) \rightarrow p(x))$$

For natural numbers:

$$\forall p : (p(1) \wedge (\forall x : p(x) \rightarrow p(x+1))) \rightarrow (\forall x \in \mathbb{N} : p(x))$$

Aside: Paradoxes can be formulated in 2nd Order Logic

"X is the set of all sets that do not contain themselves"

$$X = \{z : z \notin z\}$$

A set "is" a unary predicate: $X(z)$ holds if z is an element of X
(for example, classes, i.e., Person(x), City(x))

Logical characterization of X: $\quad X(z) \leftrightarrow \neg X(z)$,

applied to X: $\quad X(X) \leftrightarrow \neg X(X)$.

... can neither be true nor false.

How to avoid paradoxes

Paradoxes can be avoided if each variable *either* ranges over first-order things (elements of the domain) or over second-order things (predicates).

Chapter 4
RDF: Resource Description Framework

Recall (cf. Slide 37)

The *Relational Model* is a specialization of First-Order Logic:

- no function symbols. Only constant literals.
- entity-type relations:
  collect data about instances of a certain entity type,
  property names given by the attributes of the relation schema
- relationship-type relations:
  represent instances (binary or n-ary) relationship types,
  relationship name given by the table name

THE SEMANTIC WEB DATA AND KNOWLEDGE MODEL

Combination of several concepts:

- Data Model: RDF (Resource Description Framework)
  - simple logical data model
  - things: resources; Web aspect: Web-wide *Uniform Resource Identifiers*
  - statements express properties: subject-predicate-object
- Metadata: RDF Schema
  - conceptual model: classes, subclasses and properties
  - classes and properties are also resources and can be described.
    (i.e., "second-order")
  - descriptions about classes and properties allow to draw conclusions for their instances.
- database-style format + XML representation
- so far: more database-style than knowledge-base or ontology
- OWL: higher-level conceptual model
  based on description logic (a fragment of FOL), "sliced" in complexity levels

Design and Use of a Domain Ontology

Design

- define the concepts (using RDFS): names of classes, class hierarchy, properties
- define a schema for URIs
- all people then use this schema and these names

Use

- make statements with the given vocabulary about things identified by the URIs

Global Semantics

- ... just collect all statements.
- since all use the same URIs things will easily fit together.

$\Rightarrow$ The Web as a global data source

- query evaluation?
- incompleteness, inconsistencies, junk & fakes.

RDF: Resource Description Framework

RDF is another specialization of First-Order Logic without function symbols.

Extension of the ideas of conceptual modeling (ER-Model, UML) and a restricted subset of first-order logic:

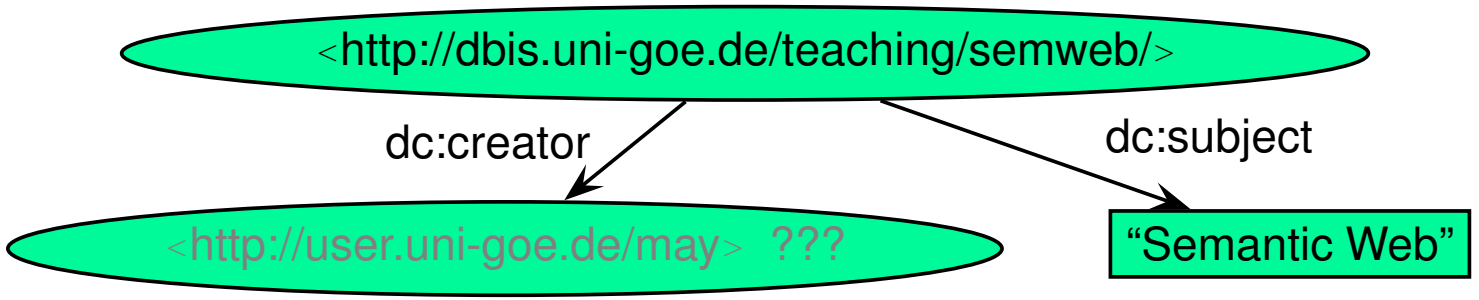
- (globally) agreed identifiers of elements of the domain as constants,
- literal constants,
- concepts (as unary predicates, e.g., Country(*germany*); usually with capital first letter),
- properties as binary relationships, e.g.,
  name(*germany*, “Germany”) , or (*germany*, name, “Germany);
  capital(*germany*,*berlin*) , or (*germany*, capital, *berlin*); usually with non-capital first letter.

Model: RDF Graph + derived knowledge

- derived knowledge: by RDFS and OWL built-in notions

4.1 RDF: Idea and Overview

- first Working Draft: 1997
- triple-based “RDF statements”: (*subject*, *predicate*, *object*)
- graphical representation ($\Rightarrow$ RDF graph), “Turtle” Unicode representation, XML representation
- *subject* and *object* are *resources* (cf. terminology for XLink)
  the object can also be a literal value (of an XML Schema datatype).
- a resource is everything ... that can be described by a URI
  (graphical notation: an ellipse)
  - a Web page, a book, a lecture ...
  - the resource identifiers are denoted with <...> to distinguish them from simple strings
  - here used with sketchy edge labels (clarified later)



RDF Model and Triples

- Initial goal: semantic description of things on the Web (i.e., Web pages and parts of them)
  author, keywords, *annotation* of links
  goal: semantics-driven Web indexing and search engines
- Information about real (Web) resources (*Triples, Statements*):
  (<http://dbis.uni-goe.de> dc:creator <http://user.uni-goe.de/may>) ???
  (<http://dbis.uni-goe.de> dc:subject "Database Group")
  (<http://dbis.uni-goe.de/teaching/semweb/> dc:type "Lecture")

Example: Dublin Core (dc)

"Dublin Core": a set of properties for describing (HTML Web) documents (defined at a metadata workshop in Dublin/Ohio 1995).

- title, creator, subject, description, date, type etc ... see
  <http://dublincore.org/documents/dces/>

RDF Model and Triples

- The Semantic Web intention was not only to annotate things, but also to *relate* them:
  (<http://dbis.uni-goe.de> uni:offers-lecture <http://dbis.uni-goe.de/teaching/semweb/>)
  (<http://dbis.uni-goe.de> dc:linksTo <http://dbis.uni-goe.de/teaching/semweb/>)
- This requires *relationships* from other domains.
- some "things" are also not real HTTP Web pages (e.g., persons)
  (the above "early" examples mix Web pages, strings, persons, and lectures in a dirty way)

$\Rightarrow$ generic notion of *resources*
(note: classes and even properties are also resources)
(note that in the graphical notation, (the usage of) properties is also depicted as labels at the edges – this *means* them as resources)

RESOURCES: URIS AND URLS

Things that are not “real” resources that have a URL (e.g. in HTTP) can get a virtual *Unified Resource Identifier (URI)*, e.g.

(<http://user.cs.uni-goe.de/∼may> dc:creator <de:person-D-12345678>)

and can then be described:

(<de:person-D-12345678> uni:position uni:Professor)
(<de:person-D-12345678> bla:lives <geo://country-de/city-goettingen>)
(<de:person-D-12345678> bla:e-mail <mailto:may@cs.uni-goe.de>)

URIs as Web-Wide Identifiers

- The URIs carry non-logical semantics as identifiers throughout the (Semantic) Web.
- information contained in the triples is independent from where they are actually (e.g., as files accessible via HTTP) located, and from their order.
- the URIs are usually be organized in *domain ontologies*.
- URIs are agreed in the same way as property names by the domain ontology designer. Real URLs and virtual URIs can be arbitrarily mixed.
- URIs to be used Web-wide can be defined freely
  - referring to the URL of the file where they are defined
    (this becomes more concrete with "Linked Open Data (LOD)" (since ∼2010); cf. Slides 286 ff.).
  - defining a URI that is completely different from the file's URL
- all members of a "community" can describe the resources in RDF.

RDF for Data Integration

- RDF files (knowledge bases) in different places that use the same URIs for "things", classes, and properties (edge labels) can be easily combined.

URIs and URLs

URI according to RFC 2396 (<http://www.ietf.org/rfc/rfc2396.txt>)

- a URI can be a locator (URL), a “name” (URN), or a “general” URI.
  General form: <*scheme*:*hierarchical_part*>
- URIs: a sequence of characters; often with a hierarchical structure
- URLs are those URIs that identify resources via their physical access mechanism (i.e., http, ftp, gopher, file, ...), in general their *schema* part names the protocol.
  <http://www.informatik.uni-goettingen.de/people.html>
  <mailto:may@informatik.uni-goettingen.de>
  <news:de.comp.text.tex>
- URI schemes can be “registered” at <http://www.IANA.org>(Internet Assigned Numbers Authority).
  example (non-registered): <isbn:3-89722-153-5>
- URIs serve as agreed *identifiers* in a certain community
- some of them are valid URLs, some of them not.

RDF uses general URIs, not only URLs.

Semantics of URIs and URLs

URIs: mainly just identifiers, can be URLs

"Typical" Objects

- Objects are resources. Their URIs can be URLs or purely virtual.
- applications that use them load some given RDF files that describe them and evaluate these files.

Properties and Classes

- They are also resources (i.e., not the names string like "City" or "hasCapital", but some URI like <geo://classes#City> and <geo://properties#hasCapital>)
  (note for later: classes and properties do not have "names", since any user-defined properties must only be applied to first-order objects)
- for querying, their URIs are just identifiers
- for an application, there *can* be an actual description of a notion at the place that is identified by the URL,
- then, an application can find this information (cf. "Linked Open Data", LOD, Slides 286 ff.)

URIs in the Semantic Web

Base URIs are often used for distinguishing ontologies

- e.g., <http://purl.org/dc/elements/1.1/> is the base URL for the Dublin Core Ontology that defines notions for annotating documents with authors etc.
- <http://www.semwebtech.org/mondial/10/> is base URI for defining the notions used in Mondial
  - entity instances are resources that have a URI in this scope:
    e.g. .../10/countries/D/provinces/Berlin/cities/Berlin
  - entity types (Country, City) and property names (capital) as e.g. .../10/meta#Country and .../10/meta#capital

Syntax and Handling of URIs and URLs

- General form: *scheme*:*hierarchical_part*
- representations (Turtle, RDF/XML) allow to use *prefixes* and a *base* (cf. XML namespaces) for shorter notation:
  - declare `prefix monmeta: <http://www.semwebtech.org/mondial/10/meta#>` `base <http://www.semwebtech.org/mondial/10/>`
  - use `monmeta:Country` as class name (stands for <http://www.semwebtech.org/mondial/10/meta#Country>)
  - use the *relative* URI `monmeta:capital` as property name
  - use `<countries/D>` (expands with the base to `<http://www.semwebtech.org/mondial/10/countries/D>`) as URI for Germany
  - use `<countries/D/provinces/Berlin/cities/Berlin>` (expands to `<http://www.semwebtech.org/mondial/10/countries/D/provinces/Berlin/cities/Berlin>`) as URI for Berlin

RDF Representations and Notations

- as triple "database": "N-triple", "Turtle" (Terse RDF Triple Language)
- graphical: as a graph
- an RDF/XML representation that represents the RDF data, used for data exchange (for every RDF database there are multiple RDF/XML serializations)
- note that although RDF data looks like a large table with three columns, it cannot be stored directly in SQL: the *subject* and *object* column must hold URIs, strings and numbers.
  - similarity with *vertically partitioned* storage of relational databases (cf. Slide 37): each predicate is stored in a binary relation.

TOOLS

The W3C RDF Validator

- `http://www.w3.org/RDF/Validator/`
- input: RDF/XML
- output: Graph and/or triples

A lightweight JENA-based locally installed tool

- JENA (`http://jena.sourceforge.net`) is a Java-based Semantic Web Framework that supports RDF/RDFS, several types of OWL reasoning, and the SPARQL language; optionally an underlying relational database system can be used.
- see Web page of the lecture for further information.

4.2 The "Turtle" RDF Notation

- Turtle is a superset of the (equivalently expressive) "N-Triple" notation, and a subset of the (more expressive) "Notation 3"/"N3" language
- Triple-"Statements" *subj pred obj .*

  (Alternatives: *subj* has *pred obj .* or *obj* is *pred* of *subj .*)
- URIs: as <*uri*>
- Literals may occur as objects: as numbers (e.g. 42) or strings (e.g., "John Doe").
- Short form for x p y_1 . and x p y_2 . : x p y_1, y_2.

  <#john> <#child> <#alice>, <#bob> .
- Short form for x p_1 x_1 . and x p_2 x_2 . : x p_1 y_1; p_2 y_2.

  <#alice> <#age> 10; <#name> "Alice" .

EXAMPLE: TURTLE

- example: use most simple “URI”s

```
<family:john> <family:name> "John";  <family:age> 35;
              <family:hasChild> <family:alice>, <family:bob> .
<family:alice> <family:name> "Alice"; <family:age> 10 .
<family:bob> <family:name> "Bob"; <family:age> 8 .
```

[Filename: RDF/family.n3]

Query language: SPARQL (details later)

```
select ?X ?N
from <file:family.n3>
where {?X <family:name> ?N}
```

[Filename: RDF/family.sparql]

- user interface of the local Jena-based tool: see Web page here: jena -q -qf family.sparql

URIs as Identifiers across RDF Files

- URIs: allow for "referencing" things across RDF files
- simplest "full" form: URNs like <schema:name>; typical for examples:

```
<family:john> <family:name> "John";  <family:age> 35;
              <family:hasChild> <family:alice>, <family:bob> .
<family:alice> <family:name> "Alice"; <family:age> 10 .
<family:bob> <family:name> "Bob"; <family:age> 8 .
```

[Filename: RDF/family.n3]

```
<family:mary> <family:name> "Mary";  <family:age> 32;
  <family:married> <family:john>;
  <family:hasChild> <family:alice>, <family:bob> .
```

[Filename: RDF/family2.n3]

```
select ?X ?C ?A
from <file:family.n3>
from <file:family2.n3>
where {?X <family:hasChild> ?C . ?C <family:age> ?A }
```

[Filename: RDF/family-both.sparql]

URIs: local identifiers

- if only a local part of the URI is given, it is by default extended with the document URL (yielding a *local* uri):

```
<#john> <#name> "John";  <#age> 35;  <#child> <#alice>, <#bob> .
<#alice> <#name> "Alice"; <#age> 10 .
<#bob> <#name> "Bob"; <#age> 8 .
```

[Filename: RDF/john-local.n3]

```
select ?X ?Y ?N
from <file:john-local.n3>
where {?X ?Y ?N}
```

[Filename: RDF/john-local.sparql]

Result (among others):

```
X / <file:///homewap1/may/teaching/SemWeb/RDF/john-local.n3#alice>
Y / <file:///homewap1/may/teaching/SemWeb/RDF/john-local.n3#name>
N / "Alice"
```

Note: when accessing the same file via

`from <https://www.dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3`

the URL expansion also uses that base URL.

URIs: relative with giving a constant @base

- if `@base <uri>` is given, relative URIs are extended with this base URI:

```
@base <foo://bla/>.
<persons/john> <meta#name> "John";  <meta#age> 35;
               <meta#child> <persons/alice>, <persons/bob> .
<persons/alice> <meta#name> "Alice"; <meta#age> 10 .
<persons/bob> <meta#name> "Bob"; <meta#age> 8 .
```

[Filename: RDF/john-base.n3]

```
select ?X ?Y ?N
from <file:john-base.n3>
where {?X ?Y ?N}
```

[Filename: RDF/john-base.sparql]

Result (among others):

```
X / <foo://bla/persons/alice>
Y / <foo://bla/meta#name>
N / "Alice"
```

- Note: hierarchically structured URLs usually begin with "//"
  (otherwise, some tools may complain)

URIs: Local Identifiers in Queries

- files with local URIs can be queried via HTTP
- combine the URI of the file + local part

```
select ?X ?N
from <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
where {?X <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#name> ?N}
```

[Filename: RDF/john-http.sparql]

- change the default expansion: use “base” for a relative addressing in the query:

```
base <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
select ?X ?Y
from <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
where {?X <#name> ?Y}
```

[Filename: RDF/john-base-local.sparql]

- result e.g., X/<http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#bob>, Y/“Bob”

Local URIs: Adding Statements to “Remote” Scopes

Even the use of *local* URIs cannot prevent others (by other RDF files) from adding statements to that scope:

```
<http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#mary>
  <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#married>
  <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3#john>.
```

[Filename: RDF/mary-remote.n3]

```
base <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
select ?X ?Y ?A
from <http://dbis.informatik.uni-goettingen.de/Teaching/SemWeb/RDF/john-local.n3>
from <file:mary-remote.n3>
where {?X <#married> ?Y . ?Y <#age> ?A}
```

[Filename: RDF/mary-http.sparql]

- note that adding resources or facts to things in “foreign” scopes is also a danger in RDF and in the Semantic Web.

$\Rightarrow$ “Trust” is an important keyword for SW applications.

USAGE OF PREFIXES

- There can be only one “base” to extend relative URIs

⇒ Usage of (multiple) “prefix” declarations (cf. namespaces in XML):

- define @prefix *pre*: <*uri*>.
  and then use expressions *pre*:*qname*
  (only qnames allowed, no hierarchical path)
  These expressions expand (roughly, see Slide 105) to <*uri qname*>.

```
@prefix : <foo://bla/meta#> .
@prefix family: <foo://bla/persons/> .
  family:john :name "John";  :age 35;  :hasChild family:alice, family:bob.
  family:alice :name "Alice"; :age 10 .
  family:bob :name "Bob"; :age 8 .
```

[Filename: RDF/john.n3]

Equivalent:

```
<foo://bla/persons/john> <foo://bla/meta#name> "John"; <foo://bla/meta#age> 35;
     <foo://bla/meta#child> <foo://bla/persons/alice>, <foo://bla/persons/bob>.
<foo://bla/persons/alice> <foo://bla/meta#name> "Alice"; <foo://bla/meta#age> 10 .
<foo://bla/persons/bob> <foo://bla/meta#name> "Bob"; <foo://bla/meta#age> 8 .
```

[Filename: RDF/john-expanded.n3]

EXPANSION OF PREFIXES

Illustrate the equivalence by stating the same query against both Turtle “fact bases”:

```
prefix : <foo://bla/meta#>    # we need only this for the :name predicate
select ?X ?N
from <file:john.n3>
where {?X :name ?N}
```

[Filename: RDF/john.sparql]

```
prefix : <foo://bla/meta#>
select ?X ?N
from <file:john-expanded.n3>
where {?X :name ?N}
```

[Filename: RDF/john-expanded.sparql]

- all positions are subject to expansion.

TURTLE PREFIX NOTATION: COMMENTS

- empty prefix: define "@prefix : *uri*" and just use ":*qname*"
- additionally, a set of non-empty prefixes can be defined as "@prefix *pre*: *uri*" and use "*pre*:*qname*"
- the results are URIs "below" the URI that is defined for the prefix
- usually, the prefix uris end with "#", or with "/"
  - "#": generates expressions like ("hash-namespace") foo://bla/meta#age (which is a property), or foo://bla/meta#Person (which is a class), or
  - "/": generates expressions like ("slash-namespace") <foo://bla/persons/alice> (which is an individual).
  - without these, there might be different behavior.
- note that after *prefix*: only qnames are allowed. Usage with paths like *pre*:$qname_1$/$qname_2$ is not allowed.
- note the similarity with the use of *namespaces* in XML:
  xmlns = *uri* use <*elementname*>
  xmlns:*ns* = *uri* use <*uri*:*elementname*>

MERGING RDF DATA

- easy Web-wide data integration when using agreed *URIs* and *notions*.
- just load several RDF files into one model: same URIs are identified.

```
@prefix : <foo://bla/meta#> . @prefix p: <foo://bla/persons/> .
p:jack :name "Jack"; :age 65; :hasChild p:john .
p:kate :name "Kate"; :hasChild p:john .
```

[Filename: RDF/parents.n3]

- foo://bla/meta# is the same URL base as earlier in john.n3:

```
prefix : <foo://bla/meta#>
select ?P ?N
from <file:john.n3>
from <file:parents.n3>
where {?X :name ?P . ?X :hasChild ?Y . ?Y :name ?N }
```

[Filename: RDF/parents.sparql]

4.3 Literals and Datatypes

- The most frequent literal types are strings (“John”) and numbers (42, 3.1415, 1.23E26) that can be represented as usual.
- Literals use *XML Schema Datatypes* (xsd:string, xsd:decimal by default)
  - Can be used in the ABox and in the TBox (as rdfs:range).
  - Further derived datatypes can be defined in OWL.
- instances of all datatypes can be represented by their *lexical representation* as string together with indicating the datatype:
- full syntax in Turtle: “*string*”^^*datatype-url*,
  - e.g. “42”^^<http://www.w3.org/2001/XMLSchema#int>
    (note: the “...” must also be present for numeric datatypes),
  - declare

    ```
    @prefix xsd:  <http://www.w3.org/2001/XMLSchema#>
    ```

    and use “1999-12-31”^^xsd:date
  - Syntax in RDF/XML:

    ```
    <mon:longitude rdf:datatype=‘‘&xsd;int’’>13</mon:longitude>
    ```

Datatypes: Date

- use string representation as in XML/XML Schema for xsd:date/time/datetime

```
@prefix : <foo://bla/meta#> . @prefix p: <foo://bla/persons/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
:birthdate rdfs:range xsd:date.
p:john a :Person; :name "John"; :age 32;
  :birthdate "1970-12-31"^^xsd:date .
p:alice a :Person; :name "Alice"; :birthdate "2000-01-01"^^xsd:date .
```

[Filename: RDF/datatype-date.n3]

- if ^^xsd:date is omitted, the ontology is detected to be inconsistent!

```
prefix : <foo://bla/meta#>   prefix p: <foo://bla/persons/>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
select ?X ?P ?Y
from <file:datatype-date.n3>
where {{p:john ?P ?Y} UNION
       {?X :birthdate ?Y . FILTER (?Y > "1999-12-31"^^xsd:date)}}
```

[RDF/datatype-date.sparql]

String Datatypes: Language Tagging

- String literals can be associated with languages, e.g. “München”@de, “Munich”@en
  A literal that has a language tag is automatically of type xsd:string.
- aside: classes and properties must not have user-defined properties; use rdfs:label etc. (cf. Annotation Properties; Slide 420)
- for functions dealing with language-tagged literals see Slide 153

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>
@base <http://www.semwebtech.org/mondial/10/>
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
<countries/D> a :Country; :name "Germany"@en, "Deutschland"@de, "ĐŞĐţÑĂĐijĐřĐ¡ĐÿÑŕ"@ru.
:Country a owl:Class; rdfs:label "Land"@de, "country"@en, "pays"@fr.
```

[Filename: RDF/language-tags.n3]

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
select ?X ?C ?N ?L ?NS ?LS  from <file:language-tags.n3>
where { ?X a ?C; :name ?N .  ?C rdfs:label ?L .
        FILTER (lang(?L) = lang(?N)) .
        BIND(str(?N) as ?NS) . BIND(str(?L) as ?LS) }
```

[Filename: RDF/language-tags.sparql]

String Datatypes: Escaping

- Escaping as usual with `"... \" ..."`, or
- using `"""` as delimiter, escaping inside is not necessary:

```
@prefix : <foo://bla/meta#> . @prefix p: <foo://bla/persons/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
:john a :Person ;
  :nickname "John \"The Hero\" Doe";
  :homepage """<ht:html xmlns:ht="http://www.w3.org/1999/xhtml">
                 <ht:body><ht:li>bla</ht:li></ht:body>
               </ht:html>"""^^rdf:XMLLiteral.
```

[Filename: RDF/string-datatypes.n3]

```
prefix : <foo://bla/meta#>
select ?X ?P ?Y
from <file:string-datatypes.n3>
where {:john ?P ?Y}
```

[Filename: RDF/string-datatypes.sparql]

Datatypes – some experiments

- it also accepts non-existing datatypes:

```
@prefix : <foo://bla/meta#> . @prefix p: <foo://bla/persons/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
p:john a :Person; :name "John"; :age "35"^^xsd:integer, "36"^^xsd:bla, "37".
p:jane a :Person; :name "Jane"; :age 35.
p:joe a :Person; :name "Joe"; :age "35.0"^^xsd:decimal.
```

[RDF/datatype-casting.n3]

- use jena -t for transform to XML/RDF.

```
prefix : <foo://bla/meta#>
select ?X ?Y
from <file:datatype-casting.n3>
where {?X :age ?Y}
```

[RDF/datatype-casting.sparql]

Y	comment
35	integer in standard notation
“36”^^<http://www.w3.org/2001/XMLSchema#bla>	
“37”	string in standard notation

- 35, "35"^^xsd:integer, and "35"^^xsd:decimal are equal (nevertheless, the output representation is that of the first binding):

```
prefix : <foo://bla/meta#>
select ?P1 ?P2 ?Y
from <file:datatype-casting.n3>
where {?P1 :age ?Y . ?P2 :age ?Y FILTER (?P2 != ?P1).}
```

[RDF/same-age.sparql]

4.4 SPARQL: An RDF Query Language

- Basis: conjunctive queries over triples,
- syntactically very similar and equivalent to Datalog queries,
- variables with multiple occurrences act as *join variables*,
- mixed with SQL-style syntax.

```
// SPARQL
select ?P ?N
from <url>                  # optional, multiple input graphs
where {?X :name ?P . ?X :hasChild ?Y . ?Y :name ?N }
```

is the same as

```
// DATALOG
?-  name(_X, P), child(_X, _Y), name(_Y, N).
```

- instead `?X`, also `$X` can be written.
- SPARQL's FROM does not correspond to SQL's selection of input relations, but to accessing multiple databases (cf. SchemaSQL)

BASIC SPARQL SYNTAX

SELECT *result variables*

FROM *input*

WHERE { dot-separated sequence of *triple-patterns* and *FILTER expressions* }

- capitalization of keywords is optional
- triple-pattern: $x\ y\ z$ as above
- FILTER expression: FILTER (*predicate expression over variables and literals*)
  (see also Functions & Operators in SPARQL)

```
prefix : <foo://bla/meta#>
select ?P ?A
from <file:john.n3>
where {?X :name ?P . ?X :age ?A . FILTER ( ?A > 30 ) }
```

[Filename: RDF/age-test.sparql]

- Object lists: x p y_1, y_2
- Predicate lists: x p_1 y_1; p_2 y_2;

SPARQL on a Database: Playing with Mondial

- above family examples: minimal toy examples
- Mondial: mondial.n3 and mondial-europe.n3 see Web page

RDF and SPARQL: Class Membership

- RDF built-in predicate: `rdf:type` for "is a" in RDF namespace
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  p:john rdf:type :Person .
- Short form: "a" (to be used without prefix!) p:john a :Person .

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>

SELECT  ?C ?N
FROM <file:mondial-europe.n3>
WHERE { ?C a mon:Country; mon:name ?N }
```

[Filename: RDF/mondial-simple.sparql]

SPARQL: Basics of Formal Semantics

The semantics of SPARQL is based on an algebra (very much like the relational algebra) for operating on *sets of answer bindings* (which actually ca be seen as tables).

- RDF Terms: ("terms" is a bad terminology; actually these are the constant symbols)
  - IRI: the set of all IRIs (Internationalized Resource Identifiers)
  - RDF-L: the set of all RDF Literals
  - RDF-B: the set of all blank nodes (see later)
  - RDF-T = IRI $\cup$ RDF-L $\cup$ RDF-B
- Var: set of variables
- *Triple Pattern* P: an element of (RDF-T $\cup$ Var) $\times$ (IRI $\cup$ Var) $\times$ (RDF-T $\cup$ Var)
  (this allows literals at subject position which does not match anything)
- *Basic Graph Pattern*: a set of triple patterns.

SPARQL: Basics of Formal Semantics (Cont'd)

... consider again BGPs over RDF-Terms and variables:

- a *mapping* μ from Var to RDF-T is a *partial* function μ : Var $\rightarrow$ RDF-T.
  (also called "(answer) substitutions" or "tuples of variable bindings"; cf. Datalog)
  for (?X mon:capital ?Y), e.g. $\mu_1 = \{X \mapsto$ <germany>$, Y \mapsto$ <berlin>$\}$
- dom$(\mu) := \{v \in$ Var : $\mu(v)$ is defined$\}$
- for a BGP P, let var$(P) \subseteq$ Var denote the set of variables that occur in P.
- for a BGP P and a mapping μ s.t. $\mathrm{dom}(\mu) \supseteq$ var(P), $\mu(P)$ denotes the replacement of every $v \in$ Var in P by $\mu(v)$.
  - $\mu(P)$ is a (ground) instance of P, i.e. a small RDF graph;
  - standard logic terminology: here the mapping is seen as a substitution (i.e., a syntactic mapping) that yields a ground instance of an expression.
- for a BGP P and an RDF graph G, an *answer substitution* is any substitution
  μ : var$(P) \rightarrow$ RDF-T such that $\mu(P) \subseteq G$
  (modulo blank node renaming, which can be done by using don't care variables in the query).
- The answer set to a BGP wrt. a graph G is the set of all answer substitutions.

SPARQL: Querying Metadata

It is totally natural in RDF and SPARQL to query also metadata:

- which properties does John have?

```
prefix : <foo://bla/meta#>
prefix p: <foo://bla/persons/>
select ?P
from <file:john.n3>
where { p:john ?P ?Y }
```

[Filename: RDF/john-properties.sparql]

- cf. SchemaSQL and F-Logic ("history" part of the SSD&XML lecture)
- especially, F-Logic (1989) had a strong influence on later languages for semistructured data, knowledge representation and the Semantic Web.

SPARQL: DISTINCT

- which properties does John have, remove duplicates?

```
prefix : <foo://bla/meta#>
prefix p: <foo://bla/persons/>
select distinct ?P
from <file:john.n3>
where { p:john ?P ?Y}
```

[Filename: RDF/john-distinct-properties.sparql]

- correctly removes duplicates.

SPARQL: OPTIONAL Conjuncts

{ ··· . OPTIONAL { *group pattern* . ··· } }
binds the variables in the inner pattern if it is satisfied:

```
prefix : <foo://bla/meta#>
SELECT ?N ?A
FROM <file:john.n3>
FROM <file:parents.n3>
WHERE { ?X :name ?N . OPTIONAL { ?X :age ?A } }
```

[Filename: RDF/optional-age.sparql]

will always bind ?N, but ?A will be bound only if the age is known.

SPARQL: OPTIONAL (Cont'd)

- note: Filter inside OPTIONAL have local scope:
  (strictly algebraic bottom-up semantics, not "sideways information passing" like in SQL correlated queries, whose algebraic translation is based on a join)

```
prefix : <foo://bla/meta#>
SELECT ?N ?A
FROM <file:john.n3>
FROM <file:parents.n3>
WHERE { ?X :name ?N .
        OPTIONAL { ?X :age ?A . FILTER (?A < 60) } }
```

[Filename: RDF/optional-age-2.sparql]

will always bind ?N, but ?A will be bound only if the age is known, and is less than 60.
If the age of ?X is known, but $\geq$ 60, the OPTIONAL part will simply bind nothing, but it will not evaluate to "false".

- How to express "list ?X if no age is known or if its age is known to be less than 60"?
  (i.e. "if the age may be less than 60")
  (see later)

SPARQL: OPTIONAL (Cont'd)

Example

OPTIONAL is bound all-or-nothing:

```
prefix : <foo://bla/meta#>
SELECT ?N1 ?N2 ?A1 ?A2
FROM <file:john.n3>
FROM <file:parents.n3>
WHERE { ?X1 :name ?N1 . ?X2 :name ?N2 . FILTER (?X1 != ?X2)
        OPTIONAL { ?X1 :age ?A1 . ?X2 :age ?A2}}
```

[Filename: RDF/age-pairs.sparql]

The result bindings contain the ages only if for *both* persons, the age is given.

SPARQL: Nested OPTIONAL

Nested OPTIONAL:

```
prefix : <foo://bla/meta#>
SELECT ?N ?A ?C
FROM <file:john.n3>
FROM <file:parents.n3>
WHERE { ?X :name ?N .
        OPTIONAL { ?X :age ?A . FILTER (?A < 60)
                   OPTIONAL { ?X :hasChild ?Y . ?Y :name ?C }}}
```

[Filename: RDF/age-and-children.sparql]

Only when an age less than 60 is given, then also the children are optionally returned.

SPARQL Filter Conditions

- atomic comparisons (=, $<$, $\leq$, $>$, $\geq$) with variables and constants as usual, string, date etc. predicates (see manual),
- logical connectives NOT ("!"), AND ("&&"), OR ("||") as usual,
- BOUND(?X) ... evaluates to true if ?X is bound in an answer,
- special predicates isIRI(?X), isLiteral(?X), etc. (see manual)

Semantics of Predicates on unbound variables

- (?X=?Y) when one of them is not bound: unknown (W3C Doc: error)

```
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
select ?N ?P
from <file:mondial-europe.n3>
where { {?C a mon:City; mon:name ?N .  OPTIONAL { ?C mon:population ?P } }
        # FILTER ((?P > 10000))   # try this also
        FILTER (!(?P > 10000)) }
```

[Filename: RDF/negatedcomparison.sparql]

- returns only those where population is known and $<$ 10000.

SPARQL: NEGATION

- !BOUND(?X) is the only form of negation in SPARQL.
- OPTIONAL together with filtering can be used for checking that a property is *not* satisfied:

```
prefix : <foo://bla/meta#>
 SELECT ?N ?A
 FROM <file:john.n3>
 FROM <file:parents.n3>
 WHERE { ?X :name ?N .
         OPTIONAL { ?X :age ?A . FILTER (?A > 60) }
         FILTER (!BOUND(?A)) }
```

[Filename: RDF/optional-age-negated.sparql]

- OPTIONAL binds ?A if an age is given and > 60, and then FILTER removes those where ?A is bound.
- people without an age or with age < 60 are returned.
- equivalent to "Negation as Failure" in Logic Programming. not bound = not found
- equivalent semantics in *defeasible logics*: "people where it is consistent to assume that their age could be < 60".

Another Example: Closed World Negation

- all countries that are not (known to be) neighbors of each other
- Close the predicate “neighbor” for the query: “neighbor does *only* hold for the pairs where it is explicitly known”.

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>

SELECT  ?C1 ?C2
FROM <file:mondial-europe.n3>
WHERE
  { ?C1 a mon:Country . ?C2 a mon:Country
 OPTIONAL
  { ?C1 mon:neighbor ?X . FILTER ( ?X = ?C2 ) }
 FILTER ( ! BOUND(?X) ) }
```

[Filename: RDF/no-neighbor1.sparql]

- The closing of the predicate is restricted to the *query*, but cannot be used in the OWL part.
- more intuitive syntactic sugar has been introduced with SPARQL 1.1 (cf. Slide 142).

Another Variant of the same Query

- all countries that are not (known to be) neighbors of each other.
- closed predicate “neighbor”: “neighbor does *only* hold for the pairs where it is explicitly known”.

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>

SELECT  ?C1 ?C2
FROM <file:mondial-europe.n3>
WHERE
  { ?C1 a mon:Country . ?C2 a mon:Country
 OPTIONAL
  { ?C1 mon:neighbor ?C2 . ?C1 mon:name ?N }
 FILTER ( ! BOUND(?N) ) }
```

[Filename: RDF/no-neighbor2.sparql]

- note: the additional property used in the OPTIONAL must be present for every ?C1 (and should be easy to compute)

EXERCISE

- For each country, give the name, and the population.
  If more than 1/4 of the population are living in its capital, give also the name and the population of the capital.
- ... to be done before discussing the formal semantics.
- Give the same queries in SQL and in XML/XQuery.

SPARQL: UNION

- { *subpattern*$_1$ } UNION { *subpattern*$_2$ }
- *subpattern*s can bind different variables (recall that in contrast, safety in the *relational calculus* requires that both disjuncts bind the same variables):

```
@prefix : <foo://bla/meta#> .        @prefix p: <foo://bla/persons/> .
 p:paul :name "Paul"; :age 30; :mother p:kate; :father p:jack.
 p:sue :name "Sue"; :age 32; :mother p:kate.
 p:peter :name "Peter"; :age 28; :father p:jack; :hasChild p:andy.
 p:andy :name "Andy"; :age 4 .  p:kate :name "Kate".
```

[Filename: RDF/father-mother.n3]

```
prefix : <foo://bla/meta#>
select ?N ?M ?MN ?F ?FN
from <file:father-mother.n3>
where { ?X :name ?N .
   {{ ?X :mother ?M . OPTIONAL { ?M :name ?MN }} UNION
    { ?X :father ?F . OPTIONAL { ?F :name ?FN }}}}
```

[Filename: RDF/father-mother.sparql]

- Exercise: if both parents are given, put both into the *same* answer.
  (Note: in the relational algebra, this is called a *full outer join*).

SPARQL: Disjunction

- Recall: logical “or” is allowed inside filters on conditions.

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?XN ?P ?A
from <file:mondial-europe.n3>
where { ?X a :Country; :name ?XN; :population ?P; :area ?A
        filter (?P > 50000000 || ?A > 400000) }
```

[Filename: RDF/or.sparql]

How to express a more complex disjunction?

Give all cities that are the capital of a country or the headquarter of an organization, or both. Give also the respective country and/or organization

```
PREFIX : <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?XN ?CC ?A
FROM <file:mondial-europe.n3>
WHERE { ?X a :City; :name ?XN
        OPTIONAL { ?C a :Country ; :carCode ?CC; :capital ?X }
        OPTIONAL { ?O :hasHeadq ?X; :abbrev ?A }
        FILTER (BOUND(?C) || BOUND(?O)) }
```

[Filename: RDF/disjunction.sparql]

SPARQL: NAMED GRAPHS

The RDF model can distinguish between stored graphs:

- Default Graph and Named Graphs
- Default Graph specified by FROM <*uri*>
- Named Graphs specified by FROM NAMED <*uri*>.
- which individuals are listed with properties in both graphs?

```
SELECT ?X ?P1 ?P2
FROM NAMED <file:parents.n3>
FROM NAMED <file:father-mother.n3>
WHERE { { GRAPH <file:parents.n3> { ?X ?P1 ?Y1 }}
        .
        { GRAPH <file:father-mother.n3> { ?X ?P2 ?Y2 }}}
```

[Filename: RDF/both-graphs.sparql]

SPARQL: Named Graphs

- Example: extract from mondial.n3 all islands in seas that are mentioned in mondial-europe.n3 (those that do not belong to european countries):

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>

SELECT DISTINCT ?I
FROM <file:mondial.n3>
FROM NAMED <file:mondial-europe.n3>
WHERE { GRAPH <file:mondial-europe.n3>
          { ?S a :Sea; :locatedIn ?C0. ?C0 a :Country .
            FILTER NOT EXISTS { ?C0 :name 'Russia' }  }
        ?I :locatedInWater ?S .
        FILTER NOT EXISTS
          { GRAPH <file:mondial-europe.n3> { ?I a :Island }}}
```

[Filename: RDF/mondial-non-european-islands.sparql]

Aside: Named Graphs and Reasoning

- Named graphs are kept separated from the default graph
- Reasoning should be applied to each separate graph
- Usually applied only to the default graph:

```
@prefix : <foo://bla/meta#> .
@prefix p: <foo://bla/persons/> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
p:john a :Person; :name "John"; :hasChild [ a :Person].
:Parent a owl:Class; owl:equivalentClass
  [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].
```

[Filename: RDF/johnP.n3]

```
@prefix : <foo://bla/meta#> .
@prefix p: <foo://bla/persons/> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
p:jack :name "Jack"; :hasChild p:john .
:Parent a owl:Class; owl:equivalentClass
  [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].
```

[Filename: RDF/parentsP.n3]

Aside: Named Graphs and Reasoning (cont'd)

```
prefix : <foo://bla/meta#>
select ?X ?C
from <file:johnP.n3>
from named <file:parentsP.n3>
where { ?X a ?C }
```

[Filename: RDF/parents-named.sparql]

SPARQL Result Modifiers

- formal semantics: yields a set of *answer substitutions* μ : var$(P) \rightarrow$ RDF-T.
- cf. SQL: SELECT *expression*, XQuery: return *expression*
- ORDER BY: SELECT ... WHERE ... ORDER BY *directives*
  where *directives* is a sequence of expressions of the forms
  - *variable*, equivalent ASC(*variable*), and
  - DESC(*variable*)
- Projection: SELECT *variables* FROM ... WHERE ...
- DISTINCT: SELECT DISTINCT *variables*
- LIMIT n: only a given number of results
- OFFSET k: start with the k-th solution
- Syntax: SELECT ... WHERE ... [ORDER BY ...] [LIMIT n] [OFFSET k].
  (note that combination of ORDER BY, LIMIT and OFFSET can be used for e.g. returning the 3rd to 6th largest items etc.)

VARIABLE ALIASING AND BINDING

- in the SELECT clause: similar as in SQL; but always in parentheses:
  (*expression* AS *variable*)

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT  ?C (?P/?A AS ?Density)
FROM <file:mondial-europe.n3>
WHERE  { ?C a mon:Country; mon:population ?P; mon:area ?A }
ORDER BY DESC(?Density)
```

[Filename: RDF/density.sparql]

- in the WHERE clause – and then use it in a FILTER:

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT  ?C ?Density
FROM <file:mondial-europe.n3>
WHERE { ?C a mon:Country; mon:population ?P; mon:area ?A .
        BIND ( ?P/?A AS ?Density )
        FILTER (?Density > 100) }
ORDER BY DESC(?Density)
```

[Filename: RDF/density2.sparql]

- A main use of this is for binding aggregated results from GROUP-BY-Subqueries (cf. Slide 146).

SPARQL Graph Constructors

- use CONSTRUCT { *dot-separated sequence of triple patterns* }
  instead of simple SELECT
- returns an RDF Graph instead of variable bindings
- result triples that contain unbound variables (e.g. due to an OPTIONAL) are not added to the graph

```
prefix : <foo://bla/meta#>
construct { ?G :grandchild ?X . ?X :name ?N}
from <file:john.n3>
from <file:parents.n3>
where {?G :hasChild ?P . ?P :hasChild ?X . ?X :name ?N}
```

[Filename: RDF/construct.sparql]

- just to test RDF/XML (see later for details):

jena -q -qf construct.sparql -ol RDF/XML -of grandchildren.rdf

SHORTCOMINGS (SPARQL 1.0)

Some can be solved by extended clause syntax (see SPARQL 1.1)

- No explicit difference/negation operation, only via OPTIONAL + FILTER + !BOUND(X),
- no grouping/aggregation.

Some are inherent to the language design

- SPARQL is not a closed language: input model (RDF) different from result model (sets of variable bindings).

$\Rightarrow$ no nested queries.

SPARQL AS A CLOSED LANGUAGE?

... would look like this:

```
prefix : <foo://bla/meta#>
select ?X
from
 { construct { ?G :grandchild ?X . ?X :name ?N}
   from <file:john.n3>
   from <file:parents.n3>
   where {?G :hasChild ?P . ?P :hasChild ?X . ?X :name ?N}
 }
where { ?X ?P ?Y }
```

[Filename: RDF/construct-nested.sparql]

- Note: this is not yet possible in standard SPARQL
- Has been done in a Master's Thesis at DBIS in 2019 by extending the JENA SPARQL module accordingly.

COMPARISON WITH MULTIDATABASES

- Multidatabases in the “old” times: a set of autonomous databases that provide correlated contents
- cf. the section on SchemaSQL in the SSD&XML lecture:
  SELECT ... FROM db_1::rel_1, db_2::rel_2, ...
  (and similar expressions)
- the SPARQL FROM clause also selects multiple input RDF data sources

Current Situation

Seeing the RDF Web as a multidatabase, RDF and SPARQL provide a unified model and language for data integration (via the URIs).

Semantic Web Vision

The actual origin of the RDF data is *not* specified by the user: users query “the Semantic Web” (via a portal) and relevant data sources will be selected transparently.

SPARQL Query Types: SELECT, CONSTRUCT, ASK and DESCRIBE Queries

- SELECT *variables* (FROM ...)* WHERE { ... } :
  "common" SPARQL query style,
  result type: sets of (partial) tuples of variable bindings
  (note: output type distinct from input type (graph(s)s)
- CONSTRUCT *pattern* (FROM ...)* WHERE { ... } : generation of triples.
  result type: set of triples / a graph
- ASK (FROM ...)* WHERE { ...} : boolean query.
- DESCRIBE *variables-or-uris* (FROM ...)* WHERE { ... }
  For URI or every variable binding, a set of triples "describing" this resource is returned.
  Details can be chosen by the service provider
  - might be all or a subset of triples that contain the resource,
  - might be an exteded set of such triples (e.g., further expanding some objects)
    (Mondial: DESCRIBE <http://.../mondial/10/countries/D>
    could deliver the graph consisting of Germany, its provinces, cities and other geographical features with all their properties)

  $\Rightarrow$ not usefully applicable with general SPARQL tools on arbitrary data sets. Mostly provided by LOD services (cf. Slides 286 ff.) for their "own" data.

4.5 SPARQL 1.1

SYNTACTIC EXTENSION: FILTER NOT EXISTS WITH SUBPATTERN

- NOT EXISTS pattern in FILTER (syntactic sugar for the !BOUND(*var)* construct)

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT  ?C
FROM <file:mondial-europe.n3>
WHERE { ?C a mon:Country
        FILTER NOT EXISTS { ?C mon:neighbor ?C2 } }
```
[RDF/mondial-not-exists.sparql]

- Recall: this is *closed-world negation*.
- Formal semantics:
  - NOT EXISTS is a boolean truth value function used *as filter*,
  - similar to the “not(...)” function in XQuery
  - some implementations accept SQL-style NOT EXISTS in a BGP without the FILTER keyword.

FILTER NOT EXISTS / MINUS

Consider again Slide 125:

- “all pairs of countries that are not (known to be) neighbors of each other”.
- The FILTER NOT EXISTS *pattern* can be used for complex negation (relational “anti-join”, including the relational difference).

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT  ?C1 ?C2
FROM <file:mondial-europe.n3>
WHERE { ?C1 a mon:Country . ?C2 a mon:Country
        FILTER NOT EXISTS { ?C1 mon:neighbor ?C2 }
      }
```

[Filename: RDF/no-neighbor3.sparql]

- equivalent:

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT  ?C1 ?C2
FROM <file:mondial-europe.n3>
WHERE { ?C1 a mon:Country . ?C2 a mon:Country
        MINUS { ?C1 mon:neighbor ?C2 }
      }
```

[Filename: RDF/no-neighbor4.sparql]

PATH EXPRESSIONS FOR NAVIGATION

- path expressions (by "/") that consist of two or more subexpressions
- ^*expr* for inverse (and *expr*^*expr* for *expr*/^*expr*),
- *expr* | *expr* for alternative,
- *expr** and *expr*+ for transitive closure, *expr*? for 0-or-1-steps,
- (JENA-ARQ-only, not SPARQL 1.1) *expr*{n,m} for n to m *expr* steps of *expr*, *expr*{n} for exactly n steps, *expr**{n,} and *expr**{,n} for at least/at most n steps of *expr*

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
# http://jena.apache.org/documentation/query/property_paths.html
SELECT  distinct ?C ?P ?O ?C2   ### what happens without "distinct" and why?
FROM <file:mondial-europe.n3>
WHERE { ?C a mon:Country ;
        mon:capital/mon:population ?P ;
        mon:capital/^mon:hasHeadq ?O ;
        # mon:neighbor+ ?C2 ;
        mon:neighbor{3,4} ?C2 . FILTER NOT EXISTS { ?C mon:neighbor{1,2} ?C2 }}
```

[Filename: RDF/mondial-path-exprs.sparql]

GROUP BY + HAVING AND AGGREGATIONS

- ... as in SQL:

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
# http://jena.apache.org/documentation/query/group-by.html

SELECT  ?C (count(?Cty) AS ?CT)
FROM <file:mondial-europe.n3>
WHERE
  { ?C a mon:Country ;
    (mon:hasCity|(mon:hasProvince/mon:hasCity)) ?Cty
  }
  GROUP BY ?C
  HAVING (count(?Cty) > 1)
```

[Filename: RDF/mondial-group-by.sparql]

SUBQUERIES IN SPARQL

- WHERE { *basic graph pattern* . { SELECT ... [FROM ...] WHERE ... }}
- cf. SQL: subqueries in the SQL FROM clause contributing to the join,
- *implicit* join variables,
- no subqueries in the FILTER (SQL's where clause).
- main use: compute some aggregate in the subquery:

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT  ?N ?MAXPOP
FROM <file:mondial.n3>
WHERE {
        ?X a mon:Country; mon:name ?N; mon:population ?MAXPOP .
        {  SELECT (MAX(?YP) AS ?MAXPOP)
           WHERE { ?Y a mon:Country; mon:population ?YP }  }
      }
```

[Filename: RDF/biggestcountry.sparql]

Example with Subquery + GROUP BY

- for each country, give its biggest city, if this has more than 1000000 inhabitants:

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?Y ?C ?CN ?MP
FROM <file:mondial.n3>
WHERE
  { ?Y a mon:Country ;  mon:carCode ?C ;
       mon:hasCity [ mon:name ?CN ; mon:population ?MP] .
   { SELECT ?Y (MAX(?YP) AS ?MP)
     WHERE {
       ?Y a mon:Country;  mon:hasCity/mon:population ?YP.
          ## short for:  mon:hasCity [ mon:population ?YP ].
       }
     GROUP BY ?Y
     HAVING (?MP > 1000000)
  }}
ORDER BY ?MP
```

[Filename: RDF/biggestcities.sparql]

MISCELLANEOUS – BUILT-IN PREDICATES ETC.

- the values from any evaluating expression can be checked for their types, e.g.,
  - isURI(*Expression*)
  - isBlank(*Expression*)
  - isLiteral(*Expression*)
  - isNumeric(*Expression*)
- "coalesce($expr_1$,...,$expr_n$)" returns the value of the first $expr_i$ that evaluates without error (unbound counts as error).
- functional "if" similar as in XQuery:
  (but: XQuery is a functional language, SPARQL is not ...)
  "if(*cond_a*, $expr_1$, $expr_2$)" for "if *cond_a* is true, then use $expr_1$, else $expr_2$",

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
SELECT  ?C ?A ?X
FROM <file:mondial-europe.n3>
WHERE { ?C a mon:Country ; mon:area ?A
        BIND (if( ?A > 100000, "big", "small") AS ?X) }
```

[Filename: RDF/silly-if.sparql]

Complex “if” example

- for each country: if it has provinces, give the biggest province, otherwise the biggest city.
- prepare both cases, choose in the “BIND”:

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?XN ?Maxthing ?Maxpop
FROM <file:mondial-europe.n3>
WHERE { ?X a :Country; :name ?XN
        OPTIONAL { SELECT ?X ?PN ?MAXPP
                   WHERE { ?X :hasProvince [ :name ?PN; :population ?MAXPP] .
                           { SELECT ?X (MAX(?PP) AS ?MAXPP)
                             WHERE { ?X a :Country; :hasProvince/:population ?PP }
                             GROUP BY ?X }}}
        { SELECT ?X ?CN ?MAXCP
          WHERE { ?X :hasCity [ :name ?CN; :population ?MAXCP] .
                  { SELECT ?X (MAX(?CP) AS ?MAXCP)
                    WHERE { ?X a :Country; :hasCity/:population ?CP }
                    GROUP BY ?X }}}
        BIND (if(bound(?PN), ?PN, ?CN) AS ?Maxthing)
        BIND (if(bound(?PN), ?MAXPP, ?MAXCP) AS ?Maxpop) }
```

[Filename: RDF/max-prov-or-city.sparql]

Aside: the same – with functional “if” in XQuery

- in XQuery as a functional language, the “if” can be done inside the term evaluations:

```
for $c in fn:doc("mondial.xml")//country
return
  <country name = "{string($c/name)}"
     maxthing = "{ if($c/province)
                   then $c/province[population[last()] =
                        max($c/province/population[last()])] /name[1]
                   else $c/city[population[last()] =
                        max($c/city/population[last()])] /name[1]
                 }"
     maxpop   = "{ if($c/province)
                     then $c/province[population[last()] =
                          max($c/province/population[last()])] /population[last()]
                     else $c/city[population[last()] =
                          max($c/city/population[last()])] /population[last()]
                 }"/>
```

[Filename: RDF/max-prov-or-city.xq]

... or ...

Aside: the same – with functional “if” in XQuery (cont’d)

- ... or with a “let”, the strategy can be similar to SPARQL:

```
for $c in fn:doc("mondial.xml")//country
let $maxprov := $c/province[population[last()] =
                  max($c/province/population[last()])],
    $maxcity := $c/city[population[last()] =
                  max($c/city/population[last()])]
return
  <country name = "{string($c/name)}"
     maxthing = "{ if($c/province)
                   then $maxprov/name[1]
                   else $maxcity/name[1]
                 }"
     maxpop   = "{ if($c/province)
                   then $maxprov/population[last()]
                   else $maxcity/population[last()]
                 }"/>
```

[Filename: RDF/max-prov-or-city2.xq]

FUNCTIONS ON LITERALS AND HANDLING OF DATATYPES

- Strings and numeric literals as usual,
- "URI" is also an atomic datatype,
- RDF Literals that are instances of other XML Schema datatypes consist of a string representation and a datatype, e.g., "1970-12-31"^^xsd:date.

There are the following functions for literals in SPARQL (to be used in filters, not directly in patterns):

- str: RDF-Literal $\rightarrow$ xsd:string – string representation/value of the literal: str("1970-12-31"^^xsd:date) = "1970-12-31".
- datatype: RDF-literal $\rightarrow$ URI: datatype("1970-12-31"^^xsd:date) = <http://www.w3.org/2001/XMLSchema#date>
- Constructor: strdt(*string*, <*uri*>) $\mapsto$ "*string*"^^<*uri*>.

SPARQL functions and operators

- often they have different syntax than in XPath/XQuery ("-" not allowed in function names)
- `https://en.wikibooks.org/wiki/SPARQL/Expressions_and_Functions`

Mathematical Operators

like in XML, in the `http://www.w3.org/2005/xpath-functions/math#` namespace:

- note: "/" is division, "*" is multiplication.
  The parser can distinguish between path expressions like :neighbor*/:name and arithmetic expressions like (?X / ?Y) * ?Z; math:pow(m, e) is m^e.

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
prefix math: <http://www.w3.org/2005/xpath-functions/math#>
SELECT ?N1 ?N2 ?Dist
FROM <file:mondial-europe.n3>
WHERE {
  ?X1 a :Country; :capital ?Y1 .
  ?Y1 :name ?N1; :latitude ?lat1; :longitude ?long1 .
  ?X2 a :Country; :capital ?Y2 .
  ?Y2 :name ?N2; :latitude ?lat2; :longitude ?long2 .
  FILTER (?N1 < ?N2)
  BIND (6370*math:acos(math:cos($lat1 / 180*3.14)*math:cos($lat2 / 180*3.14) *
         math:cos(($long1 - $long2) / 180*3.14)
        + math:sin($lat1 / 180*3.14) * math:sin($lat2 / 180*3.14)) AS ?Dist)
}
```

[Filename: RDF/distance.sparql]

Language Tagging of String Values

- String literals can be associated with languages, e.g. “München”@de, “Munich”@en
- All iana language codes are listed at `https://www.iana.org/assignments/language-subtag-registry/language-subtag-registry`

There are the following functions in SPARQL:

- str: RDF-Literal $\rightarrow$ xsd:string – string value of the literal:
  str(“München”@de) = “München”,
- lang: language-tagged-string $\rightarrow$ language tag string:
  lang(“München”@de) = “de”
- langMatches: string $\times$ language-range $\rightarrow$ {true, false}:
  checks whether a string (seen as language tags) matches a language range.
  langMatches(lang(“München”@de),“de”),
  langMatches(lang(“München”@de), lang(“Deutschland”@de)),
  usable with language-subtags: langMatches(“en-CA”, “en”),
  langMatches(lang(*literal*, “*”) is true for all literals that have a language tag.
- Constructor: strlang(*string*, *lang-tag*) $\mapsto$ “*string*”@*lang-tag*.

Functions for URIs

- uri(*string*) for creating URIs from strings.

```
prefix  mon: <http://www.semwebtech.org/mondial/10/meta#>
# http://jena.apache.org/documentation/query/select_expr.html

SELECT  ?C (?P/?A as ?D) (uri(concat(str(?CAP),'bla')) as ?U)
FROM <file:mondial-europe.n3>
WHERE
  { ?C a mon:Country ; mon:population ?P ; mon:area ?A;
    mon:capital ?CAP  }
```

[Filename: RDF/mondial-uri-fcts.sparql]

Functions for Dates

- dates and times use xsd:date
- Access functions in SPARQL: year(.), month(.) etc.
- "-" as subtraction, min/max(.) work,
- function "now()" yields the current date.

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?C ?D1 ?D2 (?D1 - ?D2 as ?X ) (YEAR(?D1) as ?Y)
where { ?C1 a :Country; :independenceDate ?D1 ; :neighbor ?C2.
        ?C2 a :Country; :independenceDate ?D2
        filter (?D1 <= ?D2)  }
```

[Filename: RDF/mondial-date-fcts1.sparql]

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select (max(?D1) as ?X) (NOW() as ?Y) (max(?D1) - NOW() as ?Z)
where { ?C1 a :Country; :independenceDate ?D1 }
```

[Filename: RDF/mondial-date-fcts2.sparql]

- note: ?Z is not bound (maybe because the indep date does not contain hours/minutes)

Constructor Functions for Typed Literals

- in RDF files: represented like “1970-12-31”^^xsd:date
- generated by the generic strdt(string,URI) constructor (Slide 151)
- constructor functions for XSD datatypes like in XPath/XQuery, e.g. xsd:date(“1970-12-31”) (in the filter, not in the pattern)

Consider the fragment from mondial using datatypes:

```
<countries/D/>  rdf:type :Country ; :name "Germany"  ;
  :independenceDate '1871-01-18'^^xsd:date  .
  :hadPopulation [ a :PopulationCount; :year "1950"^^xsd:gYear; :value 68230796].
```

Access the 1950 (year) population count of the country that became independent on 18.1.1871:

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
prefix xsd: <http://www.w3.org/2001/XMLSchema#>
select ?N ?P
where {?X a :Country; :name ?N ; :independenceDate ?I;
       :hadPopulation [ :year ?Y; :value ?P] .
       filter (?I = xsd:date("1871-01-18") && ?Y= xsd:gYear("1950")) }
```

[Filename: RDF/xsddatatypes.sparql]

4.6 SPARQL: Formal Semantics

- basically for SPARQL 1.0 without the syntactical sugar from SPARQL 1.1
- input: a graph G,
- query: consists of
  - Basic Graph Patterns (where clause) – conjunctive query (join)
  - Filters (where clause)
  - Optional (outer join)
  - Union

  not part of the algebra: result modifiers:
  - order by (evaluated after query evaluation)
  - projection (in the select clause)
  - duplicate elimination (in the select clause)
- not part of the algebra: group by and aggregations (SPARQL 1.1);
- Similarity with the relational model and its algebraic characterization.
  - actual evaluation does not use the algebra directly, but applies optimized algorithms.

Formal Semantics – Considerations

- strong similarity with relational model
- algebraic, compositional semantics desirable (cf. SQL, OQL, XQuery)
- SPARQL W3C Candidate Recommendation until v.20061004 gave a non-algebraic semantics informally in "natural language" – which is problematic for some cases.
- Strongly Recommended: Jorge Pérez, Marcelo Arenas, Claudio Gutierrez: Semantics and Complexity of SPARQL. International Semantic Web Conference 2006: 30-43 (use `http://www.dblp.org`) [PAG06]
  - develops an algebraic, compositional semantics,
  - gives a formalization of the W3C Semantics, compares both semantics.
- optional, more complex: Renzo Angles, Claudio Gutierrez: The Expressive Power of SPARQL. International Semantic Web Conference 2008: 114-129 [AG08]
- optional, lots of proofs: Jorge Pérez, Marcelo Arenas, Claudio Gutierrez: Semantics and Complexity of SPARQL. ACM Trans. on Database Systems, 34(3):16, 45 pages; 2009.
- title says it all: Jorge Pérez: Semantic Web Research Inspired by W3C Standards, or the Hell of the Practice without Theory. AMW 2012: 19 (tutorial available at `http://users.dcc.uchile.cl/~jperez/talks/amw2012-tutorial.pdf`)

Formal Semantics – Comparison with SQL

- both are basically conjunctive queries (CQs) – joins;
- SQL: base relations and views: schema: $(A_1, \ldots, A_n)$ – attributes are (usually) DB column names;
- SPARQL: tuples of variable bindings (cf. Datalog: matching variables with patterns): $(X_1, \ldots, X_n)$;
- FILTER is a selection;
- select clause (SPARQL: only finally) applies a projection;
- SPARQL: no nesting of subqueries (but nesting of {. . . }-groups with CQs, OPTIONAL and FILTER);
- OPTIONAL is an outer join (with was a rather rarely used operator in SQL)
  $\Rightarrow$ introduces lots of NULL values
  (which are much more frequent in the scenario of incomplete knowledge in the Web than in SQL)
- details of NULL values have only been sketched (pragmatically) in the DB lecture (and in the SQL lab)!

NULL VALUES IN SQL

- NULLs do not violate any integrity constraint, e.g.,

```
CREATE TABLE Country
      ( ..., population NUMBER CHECK (population >= 0), ...)
```

- note that this also holds for referential integrity constraints! e.g.

```
INSERT INTO located VALUES('Berlin',null,'A','Rhein',null,null);
```

  is not forbidden!

- NULLs do not satisfy any WHERE condition (except IS NULL)

- Join conditions in SQL are *null-rejecting*;
  a join including a null-containing column must use separate IS NULL checks or COALESCE.
  (example see next slide)

NULL Values in SQL - Join Example

```
CREATE VIEW OneProvCountry AS
SELECT country FROM province GROUP BY country HAVING COUNT(*) = 1;

CREATE TABLE city2 as
(       (SELECT name, province, country, population FROM city
         WHERE country NOT IN (SELECT country FROM OneProvCountry))
 UNION (SELECT name, NULL, country, population FROM city
         WHERE country IN (SELECT country FROM OneProvCountry)));

CREATE TABLE located2 AS
(       (SELECT city, province, country, river, lake, sea FROM located
         WHERE country NOT IN (select country from OneProvCountry))
 UNION (SELECT city, NULL, country, river, lake, sea FROM located
         WHERE country IN (SELECT country FROM OneProvCountry)));

SELECT c.name, c.population, l.river FROM city2 c, located2 l
WHERE c.name=l.city AND c.province=l.province AND c.country=l.country;
-- 958 results, the ones with NULL province are missing.

SELECT c.name, c.population, l.river FROM city2 c, located2 l
WHERE c.name=l.city
  AND COALESCE(c.province,'bla')=COALESCE(l.province,'bla')
-- note that the condition is not satisfied if one value is null, and the other is not!
  AND c.country=l.country;
-- 1157 results
```

NULL Values in SQL - IN-Example

- table “located”: (name/country/province of the city, river, lake, sea) for cities located at waters. Contains many null values.

```
select * from lake  -- 189
select * from lake where name in (select lake from located)      -- 39
select * from lake where name not in (select lake from located)  -- 0
select * from lake where name not in
              (select lake from located where lake is not null)  -- 150
```

- if a set contains “null”, every value might be equal to this “unknown” null.

4.6.1 SPARQL Algebra

(according to [PAG06])

Algebraic Syntax of SPARQL Queries

- deals with the graph pattern part, not the result modifiers (which contains projection and some operators that are also not covered by the relational algebra, like DISTINCT, ORDER BY, LIMIT, OFFSET)
- fully parenthesized
- Triple patterns (cf. Slide 115) in (RDF-T $\cup$ Var) $\times$ (IRI $\cup$ Var) $\times$ (RDF-T $\cup$ Var) are graph patterns;
- For graph patterns P_1 and P_2, the expressions (P_1 AND P_2), (P_1 OPT P_2), and (P_1 UNION P_2) are graph patterns;
- For a graph pattern P and a condition R, (P FILTER R) is a graph pattern.
  (Require filter-safeness: var$(R) \subseteq$ var(P); cf. Slide 164)
- var(P) denotes the variables occurring in a graph pattern P.

Filter-Safe Graph Patterns

Definition 4.1 ([PAG06, AG 08])

A SPARQL algebra expression is *filter-safe*, if for every subexpression of the form $(P$ FILTER $R)$, var$(R) \subseteq$ var(P). □

- For the *compositional* semantics, this requirement is absolutely necessary – what should be the semantics of {(?X p ?Y) FILTER (?Y $>$?Z)} ?
- The commonly used SPARQL pattern

```
{ ?P1 a :Person;  :age ?A1. ?P2 a :Person
  OPTIONAL { ?P2 :age ?A2 . FILTER ( ?A2 > ?A1 ) }}
```

  is not filter-safe. The filter actually formulates a join condition on the outer join:

$$(P_1 \text{ a Person; age } A_1 \text{ . } P_2 \text{ a Person}) \underset{A_1 < A_2}{\exists\!\bowtie} (P_2 \text{ age } A_2)$$

- [AG08] gives an algorithm that transforms a non-filter-safe SPARQL query into a filter-safe algebra expression.
  (Database Theory/Deductive Databases: similar to the transformation of general safe formulas into RANF when moving conjuncts into negated subformulas to make them *self-contained*; cf. exercises).
- Non-compositional sideways-information-passing evaluation is often also much more efficient, cf. evaluation of SQL vs. relational algebra.

Exercise

Recall Slide 127:

- “For each country, give the name, and the population.
  If more than 1/4 of the population are living in its capital, give also the name and the population of the capital.”
- check whether your solution is filter-safe. If not, transform it into an equivalent filter-safe query.

SPARQL Algebra: Mappings/Answer Substitutions

- a *mapping* μ from Var to RDF-T is a *partial* function μ : Var $\rightarrow$ RDF-T.
  (also called "answer substitutions" or "tuples of variable bindings"; cf. Datalog)
  for (?X mon:capital ?Y), e.g. $\mu_1 = \{X \mapsto$ <germany>$, Y \mapsto$ <berlin>$\}$
- dom$(\mu) := \{v \in$ Var : $\mu(v)$ is defined$\}$
- for a SPARQL query P, let var$(P) \subseteq$ Var denote the set of variables that occur in P.
- for a triple pattern t, and a mapping μ s.t. $\mathrm{dom}(\mu) \supseteq$ var(P), $\mu(t)$ denotes the triple obtained by replacing all variables $v \in$ Var in P by $\mu(v)$.
- two mappings μ_1 and μ_2 are *compatible* if for all $v \in$ dom$(\mu_1) \cap$ dom(μ_2), $\mu_1(v) = \mu_2(v)$, i.e., when $\mu_1 \cup \mu_2$ is also a mapping (with dom$(\mu) =$ dom$(\mu_1) \cup$ dom(μ_2)).
  - cf. definition of joining tuples $\mu_1\mu_2$ in the relational algebra,
  - here using dom(μ) (which is flexible wrt. null values) instead of the fixed schema/format of an expression.
- now consider sets Ω of such mappings (instead of sets of fixed-format tuples in the relational algebra) ...

Semantics of Expressions and Operations on Sets of Mappings

Recall: Relational Algebra defines operators π, σ, $\bowtie$ etc. on sets of tuples.

Common general notation: $[\![P]\!]$ for "semantics of P"; $[\![P]\!]_D$ for "wrt. a given graph D".

- base case: triple patterns:
  $[\![t]\!]_D = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in D\}$.
- operations on sets of mappings:
- projection: not needed here; as a result modifier, it is external to the core algebra.
- selection: next slide.
- $[\![(P_1 \text{ UNION } P_2)]\!]_D = [\![P_1]\!]_D \cup [\![P_2]\!]_D$;
- $[\![(P_1 \text{ AND } P_2)]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$ with
  $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1 \text{ and } \mu_2 \in \Omega_2 \text{ are compatible}\}$;
- $[\![(P_1 \text{ OPT } P_2)]\!]_D = [\![P_1]\!]_D \mathbin{⟕} [\![P_2]\!]_D$ with
  $\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$ where $\setminus$ is defined as
  $\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible}\}$;
  (note that "$\setminus$" here does not denote set difference, but has a different definition).

Semantics of Expressions and Operations on Sets of Mappings: Filters

Three-valued logic for evaluating conditions when variables are not bound:

- truth values t (true, 1), u (undefined, 0.5), f (false, 0), ordered by $t > u > f$.
- All three-valued logics coincide in the definition of $\wedge$, $\vee$, and $\neg$:

$A \wedge B = \min(A, B)$

A \ B	f	u	t
f	f	f	f
u	f	u	u
t	f	u	t

$A \vee B = \max(A, B)$

A \ B	f	u	t
f	f	u	t
u	u	u	t
t	t	t	t

$\neg A = 1 - a$

A	$\neg A$
f	t
u	u
t	f

- selection: $[\![(P \text{ FILTER } R)]\!]_D = \{\mu \in [\![P]\!]_D \mid \text{val}(R, \mu) = t\}$ with
  - val(bound$(?X), \mu) = (X \in \text{dom}(\mu))$ with $(X \in \text{dom}(\mu))$ as truth value $t, f \in \{t, u, f\}$;
  - $\text{val}(?X = ?Y, \mu) = \begin{cases} t & \text{if } X \in \text{dom}(\mu),\ Y \in \text{dom}(\mu), \text{ and } \mu(X) = \mu(Y) \\ f & \text{if } X \in \text{dom}(\mu),\ Y \in \text{dom}(\mu), \text{ and } \mu(X) \neq \mu(Y) \\ u & \text{if } X \notin \text{dom}(\mu) \text{ or } Y \notin \text{dom}(\mu); \end{cases}$
  - val$(?X = c, \mu)$ analogously;
  - for $(R_1 \wedge R_2)$, $(R_1 \vee R_2)$, and $(\neg R_1)$ see above truth tables.

Bag vs. Set Semantics

- SPARQL (like SQL) has a bag (=multiset) semantics.
- the above algebra (like the relational algebra) is defined wrt set semantics;
- can be generalized to bag semantics.

Exercises

1. define a relational "null-tolerant join" that acts like $\bowtie$ above.
2. which SQL construct is similar to the "$\setminus$" operator in the above SPARQL algebra?
3. in the above algebra, OPT is expressed via left outer join, which is defined via "$\setminus$" (while a corresponding MINUS does not exist in the SPARQL 1.0 syntax, it only came with SPARQL 1.1).
   Such a MINUS (cf. Exercise (2)) provides a more intuitive idea of negation than "!bound($?X$)". Give a general pattern how to express (P_1 MINUS P_2) in SPARQL syntax.
4. recall the definition of $\overline{\bowtie}$ in the relational algebra (DB lecture) and define SPARQL's $\overline{\bowtie}$ in a similar way.

Exercise

Prove or show a counterexample:

The statement (from W3C SPARQL Working Draft 20061004)

> If OPT(A, B) is an optional graph pattern, where A and B are graph patterns, then S is a solution of OPT(A,B) if
>
> - S is a pattern solution of A and of B, or
> - S is a solution to A, but not to A and B.

describes the same semantics as above.

Relationship with Datalog and the Relational Algebra

Theorem 4.1 ([AG08])

- SPARQL (=version 1.0, = the above algebra) has the same expressive power as *non-recursive safe Datalog with negation*.
  (recall: non-recursive with negation is stratifiable)
- SPARQL (=version 1.0, = the above algebra) has the same expressive power as the relational algebra (under set semantics, and under bag semantics). □

Proof ([AG08]):

- mapping from SPARQL to the filter-safe algebra expressions;
- mappings between the (filter-safe) SPARQL algebra and Datalog/relational algebra.

Normal Form

Theorem 4.2 ([PAG06])
Every graph pattern is equivalent to a pattern in the *normal form*

$$(P_1 \text{ UNION} \cdots \text{UNION } P_n)$$

where each P_i is a UNION-free graph pattern. □

Proof: UNION is distributive wrt. each of the operators:

- $(P_1$ AND $(P_2$ UNION $P_3)) \equiv (P_1$ AND $P_2)$ UNION $(P_1$ AND $P_3)$,
- $((P_1$ UNION $P_2)$ OPT $P_3) \equiv (P_1$ OPT $P_3)$ UNION $(P_2$ OPT $P_3)$,
- $((P_1$ UNION $P_2)$ FILTER $R) \equiv (P_1$ FILTER $R)$ UNION $(P_2$ FILTER $R)$,
- note that the case for $(P_1$ OPT $(P_2$ UNION $P_3))$ is harder to prove [PAG09].

Complexity

[PAG06] Consider the problem "is $\mu \in [\![P]\!]_D$"?

So-called "data complexity" for fixed P as a function of the size of D:

- The problem is in LOGSPACE (which is a subset of the complexity class P, i.e., it is polynomial).

So-called "combined complexity" as a function of P and D:

- Conjunctive queries: the problem is in $O(|P| \cdot |D|)$ for graph patterns containing only AND and FILTER. [PAG06]
- the problem is NP-complete for graph patterns containing only AND, FILTER, and UNION. [PAG06]
- the problem is PSPACE-complete for graph patterns in general. [PAG06]
- the problem is PSPACE-complete for graph patterns over OPT only. [SML10]
  [SML10]: M. Schmidt, M. Meier, and G. Lausen: Foundations of SPARQL Query Optimization. In *International Conference on Database Theory (ICDT)*, pp. 4–33. ACM, 2010.

WELL-DESIGNED GRAPH PATTERNS

Definition 4.2 ([PAG06])
A graph pattern is *well-designed* if for every subpattern $P' = (P_1 \text{ OPT } P_2)$: every variable $?X$ that occurs inside P_2 and also outside P', also occurs in P_1. □

- Some non-well-designed graph patterns exhibit weird behavior (cf. exercises).
- Note that "well-designedness" according to [PAG06] excludes negation: usage of

  `{ pattern(?X`$_1$`,...,?X`$_n$`) OPTIONAL { pattern(?X`$_{i_1}$`,...,?X`$_{i_k}$`,?Y) } }`
  `FILTER (!BOUND(?Y))`

  requires ?Y to occur inside P_2, outside P', but not inside P_1.
  ([AP11] argues that (SPARQL closed-world) negation is not appropriate for the open Web scenario with incomplete data)
- The role of well-designed patterns for SPARQL is comparable to the role of conjunctive queries for the relational algebra.

Theorem 4.3 ([PAG09])
Evaluation is coNP-complete for the fragment of SPARQL consisting of well-designed patterns. (which is significantly cheaper than PSPACE for whole SPARQL) □

Well-Designed Graph Patterns (Cont'd)

Theorem 4.4 ([PAG06])
Every *union-free* well-designed graph pattern (without filters) is equivalent to a pattern in the *normal form*

$$((\cdots(((t_1 \text{ AND } \cdots \text{ AND } t_k) \text{ OPT } O_1) \text{ OPT } O_2) \cdots) \text{ OPT } O_n)$$

where each t_i is a triple pattern and each O_j is also in this normal form. □

- + UNION: push up/out according to Theorem 4.2,
- + FILTERS: apply them as early as possible.

ASIDE: WEAK MONOTONICITY

Recall Monotonicity of queries:

- A query Q is *monotonic* if for any databases $D_1 \subseteq D_2$, $[\![Q]\!]_{D_1} \subseteq [\![Q]\!]_{D_1}$.

Example 4.1 (A monotonic query)

Q_1 = `select ?X where {?X a :Person}`

D_1 = {:john a :Person.} D_2 = {:john a :Person. :mary a :Person.}

$[\![Q_1]\!]_{D_1}$ = {{ ?X $\mapsto$:john}} $[\![Q_1]\!]_{D_2}$ = {{ ?X $\mapsto$:john}, { ?X $\mapsto$:mary}}

$[\![Q_1]\!]_{D_1} \subseteq [\![Q_1]\!]_{D_2}$. □

Example 4.2

Q_2 = `select ?X ?A where {?X a :Person OPTIONAL { ?X :age ?A}}`

D_3 = {:john a :Person; :age 35.}

Again, $[\![Q_2]\!]_{D_1}$ = {{ ?X $\mapsto$:john}}

$[\![Q_2]\!]_{D_3}$ = {{ ?X $\mapsto$:john, ?A $\mapsto$ 35}}

Here, $[\![Q_2]\!]_{D_1} \not\subseteq [\![Q_2]\!]_{D_3}$, but "nearly".

The answer tuple in $[\![Q_2]\!]_{D_3}$ *extends* the tuple in $[\![Q_2]\!]_{D_1}$ - "more information" □

Weak Monotonicity (cont'd)

[AP11] Marcelo Arenas, Jorge Pérez: Querying Semantic Web data with SPARQL. PODS'11

Definition 4.3 ([AP11])

- For mappings μ_1 and μ_2, μ_1 *is subsumed by* μ_2, denoted by $\mu_1 \preceq \mu_2$, if $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ and $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1)$.
- For sets Ω_1 and Ω_2 of mappings, Ω_1 *is subsumed by* Ω_2, denoted by $\Omega_1 \sqsubseteq \Omega_2$, if for every for every $\mu_1 \in \Omega_1$, there exists $\mu_2 \in \Omega_2$ such that $\mu_1 \preceq \mu_2$.
- A SPARQL graph pattern P is said to be weakly monotonic if for every pair G_1, G_2 of RDF graphs such that $G_1 \subseteq G_2$, it holds that $[\![P]\!]_{G_1} \sqsubseteq [\![P]\!]_{G_2}$. □

The above Q_2 is weakly monotonic.

Theorem 4.5 ([AP11])
Every well-designed graph pattern is weakly monotonic. □

- the converse direction is an open question, but is expected to hold.

Some Observations

- Recall: Closed-World negation is nonmonotonic
  (additional knowledge may invalidate before conclusions)
- Recall: Open-World negation is monotonic
  (learning something that was not known to hold or not to hold before may not invalidate any conclusion made before).
- Recall: Default reasoning (like “normally ... holds”) is nonmonotonic
  (learning that something is an exeption from normality may invalidate some conclusions made before).
- weak monotonicity itself excludes that any conclusion made before may be invalidated.

Some Observations (cont'd)

Consider queries Q with an arbitrary `construct` clause instead of a `select` clause (using the same variables):

- for every input RDF graph G, their result $[\![Q]\!]_G^{construct}$ is an RDF graph,
- $[\![Q]\!]_{G_1} \sqsubseteq [\![Q]\!]_{G_2}$ implies $[\![Q]\!]_{G_1}^{construct} \subseteq [\![Q]\!]_{G_2}^{construct}$.

$\Rightarrow$ weak monotonicity wrt. sets of tuples of variable bindings implies monotonicity wrt. the constructed graph.

$\Rightarrow$ the definition of "weak monotonicity" is only necessary because of the (relational-style) output format of SPARQL queries.

- from the informational point of view, weak monotonicity means monotonicity.

BACK TO "WELL-DESIGNED" QUERIES

Theorem 4.5 then reads as

- Every well-designed graph pattern with `construct` output is monotonic.
  (which is rather obvious from its normal form)
- if the converse direction –which is expected to hold– holds, every monotonic SPARQL graph mapping can be expressed in the above normal form.

4.7 RDF vs. Object Identity

- So far, RDF is just a restricted variant of ground (sorted) relational First-Order Logic atoms:
  (cf. Slides 217 for a more detailed consideration)
  - only binary predicates
  - literal values and relationships carry semantics
  - URIs as constant symbols
  - no function symbols other than constants
- Sorted domain: URIs and Literals

Existential Knowledge

Sometimes the existence of things is relevant, without actually naming and giving an ID to an object.

- “A parent is a person that has at least one child”
- “John has a child of 12 years”

BLANK NODES – EXISTENTIAL QUANTIFICATION

- Short form for x p y_1 ., y_1 q_1 z_1 . and x p y_2 ., y_2 q_2 z_2 . when the URIs for y_1 and y_2 are not relevant (purely existential "blank nodes"):
  x p [q_1 z_1], [q_2 z_2] .

  <#john><#name> "John"; <#age> 35 ;

  <#child> [<#name> "Alice"; <#age> 10] , [<#name> "Bob"; <#age> 8] , [<#age> 12] .

- note that also #john is just an identifier that contains no actual information (we could also have chosen #pers123 instead). It can also be replaced by using a blank node:

  [<#name> "John"; <#age> 35;

  <#child> [<#name> "Alice"; <#age> 10] , [<#name> "Bob"; <#age> 8] , [<#age> 12]] .

RDF "Model"

Equivalent expression in FOL:

$$\exists j : (\text{name}(j, \text{“John”}) \wedge \text{age}(j, 35) \wedge$$
$$\wedge \exists a, b, c : (\text{child}(j, a) \wedge \text{child}(j, b) \wedge \text{child}(j, c) \wedge$$
$$\wedge \text{name}(a, \text{“Alice”}) \wedge \text{age}(a, 10) \wedge \text{name}(b, \text{“Bob”}) \wedge \text{age}(b, 8) \wedge \text{age}(c, 12)))$$

Example: Blank Nodes

```
@prefix : <foo://bla/meta#>.
 [ :name "John";  :age 35;
   :hasChild [:name "Alice"; :age 10] ,
               [:name "Bob"; :age 8] , [:age 12 ]] .
```

[Filename: RDF/john-blank.n3]

```
prefix : <foo://bla/meta#>
select ?X ?N
from <file:john-blank.n3>
where {?X :name ?N ; :hasChild [:age 12] }
```

[Filename: RDF/john-blank.sparql]

- generated by “[...]” above: “something that satisfies”
- nested “term” syntax for implicit conjunction of atoms, without using identifiers or key references.
  [Note that F-Logic terms (1989) and JSON expressions (2006, 2013) have a very similar structure]

Example:

Extend the database such that John is married to Mary, who is the mother of both children.

NAMED BLANK NODES

- blank nodes can also be named by local identifiers of the form "_:*localname*" that allow for "referencing" things again (existential variables).

```
@prefix : <foo://bla/meta#>.
  [ :name "John";  :age 35;
    :hasChild _:a, _:b ;
    :married [ :name "Mary";  :hasChild  _:a, _:b ] ] .
  _:a :name "Alice"; :age 10 .
  _:b :name "Bob"; :age 8 .
```

[Filename: RDF/john-married.n3]

```
prefix : <foo://bla/meta#>
select ?N ?C ?A
from <file:john-married.n3>
where {?X :hasChild ?C . ?X :name ?N . ?C :age ?A }
```

[Filename: RDF/john-married.sparql]

PITFALLS OF EXISTENTIAL KNOWLEDGE

Consider again John's family and its logical formalization (Slides 182 and 181).

```
@prefix : <foo://bla/meta#>.
 [ :name "John";  :age 35;
   :hasChild [:name "Alice"; :age 10] ,
               [:name "Bob"; :age 8] , [:age 12 ]] .
```

[Filename: RDF/john-blank.n3]

How many children does John have?

- what does a human reader understand?
- is this entailed by the logical formalization?

Pitfalls of Existential Knowledge (Cont'd)

- Alice:

```
[ :name "John"; :hasChild [:name "Alice"]]
```

- His health insurance:

```
[ :name "John"; :hasChild [:age 10]]
```

- Grandmother:

```
[ :name "John"; :hasChild [:nickname "My sunshine"] ]
```

- A neighbor:

```
[ :name "John"; :hasChild [:nickname "The little beast"] ]
```

  How many children does John have?

  – merging this yields one RDF graph.

  – the *logical formalization* is very similar to the one from the previous slide.

Pitfalls of Existential Knowledge (Cont'd)

- from the first example, most people conclude that John has three children.
  (note that "at least three" would be more exact)
- from the second example (text) many people conclude that all statements describe Alice
- the RDF graph of the second example, most people conclude that there are four children
- formally in all cases: the knowledge bases entail only that John has at least one child!
- for the first example:
  - there is a logical model where there is only one child with three names and a property "age" that has three values.
  - people use meta knowledge that age and name are functional properties. This is not contained in the RDF data!
- OWL allows to express such additional information.
- the next slides anticipate some OWL stuff that will be discussed in detail later.

Pitfalls of Existential Knowledge (Cont'd)

Show that it is consistent to assume that John has exactly one child:

- define a class OneChildParent that restricts the cardinality of child to one and make John an instance of it (using OWL; see later):

```
@prefix : <foo://bla/meta#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
 [ :name "John";  :age 35;
   :hasChild [:name "Alice"; :age 10], [:name "Bob"; :age 8], [:age 12 ];
   rdf:type :OneChildParent].
:OneChildParent owl:equivalentClass [rdf:type owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 1].
```

[Filename: RDF/john-silly-example.n3]

- how old is Alice (invoke with -pellet)?

```
prefix : <foo://bla/meta#>
select ?X ?A  from <file:john-silly-example.n3>
where {?X :name "Alice" . ?X :age ?A}
```

[Filename: RDF/john-silly-example.sparql]

Pitfalls of Existential Knowledge (Cont'd)

- Assert that age is a functional property:

```
@prefix : <foo://bla/meta#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
 [ :name "John";  :age 35;
   :hasChild [:name "Alice"; :age 10], [:name "Bob"; :age 8], [:age 12 ];
   rdf:type :OneChildParent].
:OneChildParent owl:equivalentClass [rdf:type owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 1].
:age rdf:type owl:FunctionalProperty.
```

[Filename: RDF/john-silly-example-2.n3]

```
prefix : <foo://bla/meta#>
select ?X ?A  from <file:john-silly-example-2.n3>
where {?X :name "Alice" . ?X :age ?A}
```

[Filename: RDF/john-silly-example-2.sparql]

- pellet complains about an inconsistent ontology.

Aside: Reasoning under Inconcistency

Consider the situation of the previous slide.

- Semantics of boolean implication:
  Any formula of the form false $\rightarrow \varphi$ is true in any interpretation (equivalent to $\neg$false $\vee \varphi$, which is in turn equivalent to true $\vee \varphi$.
- Given an inconsistent ontology formalization Φ, for *every* formula φ, $\Phi \models \varphi$ holds:
  for every model $\mathcal{M}$ such that $\mathcal{M} \models \Phi$ (note that there are no such $\mathcal{M}$), also $\mathcal{M} \models \varphi$ holds.
- Tableau Calculus: Given an ontology formalization Φ,
  prove $\Phi \models \varphi$ by starting a tableau over $\Phi \wedge \neg\varphi$ and trying to close it.
  If already Φ is inconsistent, any tableau over Φ can be closed without using $\neg\varphi$. Thus, $\Phi \vdash_{\text{Tableau}} \varphi$.

For that reason, it is reasonable that a prover even does not start working with an inconsistent ontology.

- research issue: reduce an inconsistent ontology to a conistent subset and prove or refute φ from this.

Pitfalls of Existential Knowledge (Cont'd)

- Assert that age is a functional property and look what is entailed ...

```
@prefix : <foo://bla/meta#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
 [ :name "John";  :age 35;
   :hasChild [:name "Alice"; :age 10], [:name "Bob"; :age 8], [:age 12 ]].
:OneChildParent  owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 1].
:TwoChildrenParent  owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 2].
:ThreeChildrenParent  owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :hasChild; owl:minCardinality 3].
:age rdf:type owl:FunctionalProperty.
```

[Filename: RDF/john-three-children.n3]

(continue next slide)

Pitfalls of Existential Knowledge (Cont'd)

```
prefix : <foo://bla/meta#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X ?C
from <file:john-three-children.n3>
where { ?X :name "John" . ?X rdf:type ?C }
```

[Filename: RDF/john-three-children.sparql]

- SPARQL counts the bindings:

```
prefix : <foo://bla/meta#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X count(?C)
from <file:john-blank.n3>
where { ?X :name "John"; :hasChild ?C }
group by ?X
```

[Filename: RDF/john-count.sparql]

- count(?C) is 3.
  SPARQL semantics is closed-world, and "unique blank node assumption"

UNIQUE NAME ASSUMPTION (DATALOG)

Datalog makes the *Unique Name Assumption*: different constants denote different things, e.g.

$$\text{sibling}(X, Y) \text{ :- } \text{child}(\_Z, X),\ \text{child}(\_Z, Y),\ X \neq Y.$$

will derive sibling(alice,bob) when facts child(john,alice) and child(john,bob) are given.

NO UNIQUE NAME ASSUMPTION IN RDF

- RDF model theory: objects described by different URIs may be the same!
- even if the children are explicitly described by URIs, the consequences are the same as in the previous example,
- in OWL, it can be explicitly stated that two URIs denote the same or different things.
- equivalence: e.g. used in ontology mapping.
- if no such information is explicitly given, both is assumed to be possible.
- equality or non-equality can be derived (e.g., by declaring a property to be functional) (Note that the same is the case for F-Logic model theory with functional methods.)

ASIDE: SOME THEORY

Consider the Turtle fragment consisting of expressions as follows:

- no explicit subjects,
- no URIs at object positions (only literals and [...] expressions),
- no named blank nodes,
- i.e., constants are only literals and property name URIs.

Expressiveness:

- Only existentially quantified variables.
- can only represent tree-like relationship structures. No cycles (cf. the “mary” example).
- The expressiveness is exactly that of FOL formulas over an alphabet of only two variables. (note that these can be used several times)

Exercise: express the RDF/Turtle “database” given on Slide 181 by a first-order formula that uses only two variables.

SOME THEORY (CONT'D)

- FOL with two variables is decidable,
- satisfiability of first-order formulas that use three variables is in general undecidable ("3-SAT")!
- restriction to two variables: guarantees a tree-like "structure" of the models ("tree model property"). This locality guarantees decidability.
- for the same reason, also many variants of Description Logics (see later) are decidable. Proofs use the equivalence to modal logics over tree-like Kripke Structures.

This RDF fragment will be relevant later when considering decidable fragments of Description Logics and OWL.

SEMANTICS: NEGATION

- compare the FOL axiomatizations of John and his family – they always describe only the things that are there, and say nothing about the (non)-existence of other things:
  Has John a child of the age of 14?

  The database does not find any, but the formula

  $$\exists j : (... \wedge \exists d : \mathsf{child}(j, d) \wedge \mathsf{age}(d, 14))$$

  is consistent – thus it has models. It is actually possible that John has such a child which is simply unknown to the database ("Open-World-Assumption" (OWA)).
- another RDF Web source could add RDF triples that describe additional children of John.
- Minimal-Model-based reasoning or SQL databases would answer "no"
  "Closed-World-Assumption" (CWA).
- SPARQL has CWA; it would find (and count) three children of John
  (kind of "unique blank node assumption")
- in general: OWA is more appropriate to the Web.
- so far: SPARQL does only matching on an RDF graph. Later: RDF/RDFS/OWL *model* that extends the graph by further knowledge that is also presented in RDF format.

4.8 RDF Conceptual Model and Built-in Predicates

... so far, RDF defines just an edge-labeled graph.

RDF provides also a simple type system (which is exteded later by RDF Schema):

- the RDF conceptual model knows about class/type membership and properties,
- rdf:type: is a special property that assigns a type (means: class membership) to something

Types and Properties as Resources

Extend the “Persons” running example from Slide 103:

```
@prefix : <foo://bla/meta#> .
@prefix p: <foo://bla/persons/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
  p:john rdf:type :Person; :name "John";  :age 35;
         :hasChild p:alice, p:bob.
  p:alice :name "Alice"; :age 10 .
  p:bob :name "Bob"; :age 8 .
```

[Filename: RDF/john.n3]

- <foo://bla/persons/john> (and others) are the URIs of objects that are described by the database,
- <foo://bla/meta#Person> is the URI of the class “Person”
- <foo://bla/meta#name> and <foo://bla/meta#age> are the URIs of the property names.

Design *ontologies* in RDF like this.

ONTOLOGY DESIGN

- separate part for “notions” (often as a “#”-namespace)
- separate part for objects (either as “#”-namespace or as “/”-namespace)
- note that after *prefix*: only qnames are allowed. Usage with paths like *pre*:$qname_1$/$qname_2$ is not allowed.
- [later in RDF/XML xml:base can be used together with paths]

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix : <foo://bla/meta#> .
@prefix persons: <foo://bla/persons/> .
@prefix cats: <foo://bla/cats/> .
 persons:john rdf:type :Person; :name "John"; :age 35;
    :has_cat cats:garfield.
 cats:garfield rdf:type :Cat; :name "Garfield"; :age 4 .
```

[Filename: RDF/john-and-the-cat.n3]

- There are <foo://bla/persons/john> and <foo://bla/cats/garfield>.

ONTOLOGY DESIGN AND QUERY FORMULATION

```
prefix terms: <foo://bla/meta#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X ?T ?A
from <file:john-and-the-cat.n3>
where { ?X rdf:type ?T . ?X terms:age ?A }
```

[Filename: RDF/john-and-the-cat.sparql]

- note that most queries only need the prefix for the metadata
- the prefix/base for data is only used when a query explicitly accesses an object by its URI

```
base <foo://bla/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?T ?A
from <file:john-and-the-cat.n3>
where { <persons/john> rdf:type ?T ; <meta#age> ?A }
```

[Filename: RDF/sparql-base.sparql]

- Alternative: value-based selection of an object

```
... where { [ :name "John" ; rdf:type ?T; :age ?A ] }
```

METADATA AS RESOURCES

... on the way to more expressive ontology languages:

- Some metadata notions are defined in the rdf namespace
  - @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  - rdf:type as a property: `:john rdf:type :Person`
    (short syntax in Turtle: `:john a :Person`)
  - rdf:Property as a class: `:hasChild rdf:type rdf:Property`
  - rdfs:Class (and owl:Class) as a class: `:Person rdf:type rdfs:Class`
- transition/not-so-well-defined language border to RDF Schema (RDFS)
- RDF tools know about rdf:type, rdf:Property, rdfs:Class etc.

note: rdf:type non-capitalized (is a property), rdf:Property (is a type) capitalized.

TYPES AND PROPERTIES

- these things are built-in notions that have in pure RDF still no meaning.
- Types are *not* XML Schema complex types (structural typing), but *semantical types*.
- Types/classes and properties are also resources that can be described this way.
- nevertheless, in pure RDF there is no kind of signature or schema; especially also no type constraints on properties.
- it's just data.

RDF Instance Data and Metadata Examples

- Consider
  reasonable: (http://.../geo#germany rdf:type http://.../geo#Country)
  reasonable: (http://.../geo#berlin rdf:type http://.../geo#City)
  non-reasonable: (http://.../geo#germany rdf:type http://.../geo#berlin)
  $\Rightarrow$ things that are of some type should not be types themselves?
- but this conflicts with meta-metadata:
  reasonable: (http://.../bio#Penguin rdf:type http://.../bio#Species)
  reasonable: (http://.../bio#tweety rdf:type http://.../bio#Penguin)
- Consider
  reasonable: (http://...#has_capital rdf:type rdf:Property)
  not reasonable: (http://...#has_capital http://...#has_population 42)
  $\Rightarrow$ a property does not have other properties?
- RDFS provides specific built-ins for metadata:
  reasonable: (http://.../geo#River rdfs:subClassOf http://.../geo#Water)
  reasonable: (http://.../geo#has_capital rdf:type rdf:Property)
  reasonable: (http://.../geo#has_capital rdfs:range http://.../geo#City)

$\Rightarrow$ these "legal" and "non-legal" usages are not enforced by anything. Non-legal usage can lead to severe problems (see Slide 228).

4.9 Example: Mondial in RDF

- RDF level: structure of URIs, types and relationships
- later: RDFS and OWL add further metadata information and knowledge

Design of the Mondial Ontology

- we chose classes and properties of the Mondial ontology to be identified by

  http://www.semwebtech.org/mondial/10/meta#*classname*

- the URIs for identifying countries, organizations etc. are actual URLs formed like

  http://www.semwebtech.org/mondial/10/countries/*code*
  http://www.semwebtech.org/mondial/10/countries/*code*/cities/*name*

- Alternatives will be discussed later (with RDF/XML).

EXAMPLE FRAGMENT

```
@prefix monmeta: <http://www.semwebtech.org/mondial/10/meta#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
<http://www.semwebtech.org/mondial/10/countries/D>
  rdf:type monmeta:Country ;
  monmeta:name "Germany" ;
  monmeta:code "D" ;
  monmeta:population 83536115 ;
  monmeta:capital
    <http://www.semwebtech.org/mondial/10/countries/D/provinces/Berlin/cities/Berlin>.
```

[Filename: RDF/a-bit-mondial.n3]

- Mondial is available in Turtle format on the Web site

- the type “Country” is <http://www.semwebtech.org/mondial/10/meta#Country>
- the property “capital” is <http://www.semwebtech.org/mondial/10/meta#capital>
- use
  @base <http://www.semwebtech.org/mondial/10/> for the individuals’ URIs.

```
base <http://www.semwebtech.org/mondial/10/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?Y
from <file:a-bit-mondial.n3>
where { <countries/D> rdf:type ?Y }
```

[Filename: RDF/a-bit-mondial.sparql]

Relationship with XML Namespaces

Consider the following XML fragment:

```
<mon:mondial xmlns:mon="http://www.semwebtech.org/mondial/10/meta#">
  <mon:Country car_code= "D">
    <mon:name>Germany</mon:name>
  </mon:Country>
</mon:mondial>
```

There's more behind these namespaces, which is not used in plain XML.

See later with RDF/XML.

RDF Summary

- more or less simple data model,
- possibility to use property and class names that are agreed Web-wide,
- the idea of URIs,
- the contents of RDF files can be integrated in a natural way via URIs and names.

4.10 Further Topics

- some additional RDF syntax: Reification and Collections. Later.
- Schema information: RDF Schema
- RDF/RDFS Model Theory + Reasoning
- More than schema information: Ontology specification by Description Logics and OWL
- How to provide RDF data and metadata on the Web?
  Later: RDF/XML. It's just another representation of RDF, RDF Schema and OWL data in a special XML syntax.
- What's missing?
- Rules?

Chapter 5
RDF Schema

Schema Information and Reasoning in an Open World

ONTOLOGIES

Schema languages, metadata languages, modeling languages, ontologies ...

Classical Data Models: seen as Specification and Constraints

- every schema description defines a (more or less complete) ontology:
- ER Model (1976, entity types, attributes, relationships with cardinalities),
- UML (1997, classes with subclasses, associations with cardinalities, OCL assertions to schema components etc.).

Knowledge Representation

Metadata provides additional information about resources of a type, or about a property.

- F-Logic signatures (1989),
- ... RDFS and OWL (Web Ontology Language)

Schema Information in an Open World

- schema describes
  - allowed properties for an object,
  - datatype constraints for literal properties [Here: XSD literal types],
  - allowed types/classes for reference properties,
  - cardinality constraints.

Closed World: Schema as Constraints

- a database must satisfy the constraints. It must be a *model* of the formulas – *the given data alone must be a model*.

Open World: potentially incomplete knowledge

- schema information as *additional information*
- since the world must be a model of the schema, some information can be *derived* from the schema.
- complain only if information is *contradictory* to the schema.

METADATA INFORMATION: TYPES, PROPERTIES, AND ONTOLOGIES

- Types and properties (i.e., everything that is used in a namespace) are not only “names”, but are resources “somewhere in the Web”, identified by a URI (used in RDF or in XML via namespaces).

$\Rightarrow$ a *domain ontology* describes the notions used in a namespace.

Schema and Ontology Information

- what types/classes are there,
- subclass information,
- what properties objects of a given type must/can have,
- to what types some property is applicable and what range it has,
- cardinalities of properties,
- default values,
- that some properties are transitive, symmetric, subproperties of another or excluding each other etc.

5.1 RDF Schema Notions - Overview

- RDF is the instance level
- cf. XML: DTDs and XML Schema for describing the structure/schema of the instance (DTD: no atomic datatypes, only tree structure; XSD has atomic datatypes)
- RDF Schema: stronger than DTD/XML – "semantic-level"
  - describe the structure of the RDF instance (i.e. the "schema" of the RDF graph, not of the RDF/XML file):
  - describes the schema *semantically* in terms of an (lightweight) ontology (OWL provides then much more features):
    - class/subclass
    - property/subproperty, domains and ranges
  - atomic datatypes for literal properties.

PREDEFINED RDFS CLASSES

The obvious ones

rdfs:Resource is "everything". All things described by RDF are called resources, and are instances of the class rdfs:Resource. This is the class of everything. All other classes are subclasses of this class. rdfs:Resource is an instance of rdfs:Class.

rdfs:Class : all things (resources and literals) are of rdf:type of some rdfs:Class. rdf:Properties have an rdfs:Class as domain and another rdfs:Class or rdfs:Datatype as range.

mon:Country rdf:type rdfs:Class.

An rdfs:Class is simply a resource X that is of (X rdf:type rdfs:Class). Usually, class names start with a capital letter.

Later, **owl:Class** will provide more interesting concepts of *intensionally defined* classes – like "the class father is the class of things that are male and have children".

rdf:Property is a subset of rdfs:Resource that contains all properties.

mon:capital rdf:type rdf:Property.

Usually, property names start with a non-capital letter.

[note: it's rdf:Property, not rdfs:Property!]

PREDEFINED RDFS CLASSES

rdfs:Datatype is the class of datatypes.

rdfs:Literal is the subclass of rdfs:Resource that contains all literals (i.e., values of rdfs:Datatypes).
Literals do (usually) not have a URI, but a literal representation (as already discussed for integers and strings).

E.g. the following holds

@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
xsd:int rdf:type rdfs:Datatype .

- Note that *reification* takes place here: rdfs:Datatype is both an instance of and a subclass of rdfs:Class! Each instance of rdfs:Datatype is a subclass of rdfs:Literal.

PROPERTIES (OF CLASSES AND PROPERTIES) IN THE RDFS VOCABULARY

rdfs:subClassOf specifies that one rdfs:Class is an rdfs:subClassOf another:

:Cat rdfs:subClassOf :Animal .

rdfs:subPropertyOf specifies that one rdf:Property is an rdfs:subPropertyOf another:

:hasCat rdfs:subPropertyOf :hasAnimal .

rdfs:domain specifies that the domain of an rdf:Property is a certain rdfs:Class:

:hasCat rdfs:domain :Person .

(does not mean that every person has a cat, but everything that has a cat is a person)

rdfs:range specifies that the range of an rdf:Property is a certain rdfs:Class
(note that rdfs:Datatype is a subclass (and an instance) of rdfs:Class):

:hasCat rdfs:range :Cat .
:age rdfs:range xsd:int .

That means, considering the (s,p,o) triples,
rdfs:domain asserts the classes of the resources in the “s” position, and
rdfs:range asserts the classes of the resources in the “o” position.
(note that it is not required to the most specific classes – owl:Thing as rdfs:domain and rdfs:range is always correct)

INFERENCE RULES

- until now, the SPARQL query language was applied to pure RDF facts (*extensional knowledge*)
- RDFS adds *inference rules* (= *intensional knowledge*)
- Queries are then not evaluated against the *fact base*, but against the *model* of the factbase and the rules.
- for this, a *reasoner* is required.

$\Rightarrow$ underlying *entailment relationship* based on *model theory*.

  - obviously, straightforward first-order logic model theory is not appropriate:
  - talk not only about elements of the domain, but also about predicates (classes and properties),
  - but needs only simple conjunctive+union queries (SPARQL) and the above-mentioned RDFS expressiveness.

5.2 RDF/RDFS Model Theory

This section gives insights in the scientific background of defining new theoretical and practical frameworks in general:

- intution: “I want to express .... and it should mean ... and it should work like this ...”
- theory: show that it works by formalizing it, investigating it, analyzing border cases and limits.
  Decidability and complexity results, calculi/algorithms,
  leads to implementations.
- User’s perspective: grasp the idea, get documentation, get implementation, press button.
  But depending on the expressiveness/complicatedness the user sometimes needs some deeper intuitive understanding of the dirty details.

  Example: what does SQL do with

```
select name
from river
where name not in (select river from located)
```

  when located.river can hold null values?
  (the answer set is empty, because null “could be” “this” river.name)

5.2.1 "Model Theory"

An important notion from mathematical logic:

- Whenever defining a (logical) framework, its syntax and its "meaning/semantics", i.e. the meaning of *truth* in it, must be formally defined.
- Decidability and complexity issues are investigated based on the model theory.
- usually, it defines an *entailment* relation $\varphi \models \psi$
- E.g., first-order logic model theory is taught in the "Formal Systems" lecture; it is based on the notion of *first-order interpretations*.
- Implementations (e.g. algorithms for symbolic reasoning) must be correct (and preferably complete).
- In some cases, the model theory of a framework can be mapped onto another, existing model theory (then reusing theorems, proof, algorithms etc.)

Different Model Theories: Examples – Overview

- E.g., first-order logic model theory is taught in the "Formal Systems" lecture; it is based on the notion of *first-order interpretations*: Entailment: $\varphi \models \psi$ if in all models of φ, also ψ holds.
  It is in general undecidable, reasoners (based on calculi) are correct, but incomplete.
- SQL uses a different model theory: closed-world negation.
  Its model theory is subject of the "Database Theory"/"Deductive Databases" lecture: the Datalog language(s), Minimal Model, Stratified Model, Well-Founded Model, Stable Models.
  While MinM and StratM are unique and total for every Datalog knowledge base, the WFM is unique, but can be partial, and there may be zero to *many* stable models (covering disjunction).
  Entailment: "cautious reasoning" - what holds in all stable models, "credulous reasoning": what holds in some stable model ("possible").
  Decidable; MinM, StratM, WFM polynomial (!), stable models exponential.
- Default Logics etc: handling sentences like "normally, birds fly":
  "Preferential models": the "most normal" model(s). The more "normal" a model, the more "preferred" is it. Heavily undecidable, and even in "commonsense cases" high complexity.
- Modal Logics (e.g., linear or branching temporal logics, logics of knowledge and belief) have their own model theories modeling many states (usually based on propositional or FOL).
- Simple case FOL+Equality: in RDF, URIs are in the domain, and there can be *equality* between them. Equality is already not a "normal" binary predicate in FOL, but has built-in axioms: it is symmetric, transitive, and whenever $c = d$ holds, these can be replaced by each other everywhere.

FOL Model Theory does not (directly) cover RDFS Model Theory

- FOL: Recall the definition of a FOL *interpretations*: constant symbols are mapped to the domain, while predicate symbols are mapped to relations over the domain, i.e. they are *not* themselves in the domain.
- In RDF, predicates (predicate symbols) are also objects of discourse.
  [note that classes are just unary predicates while properties are binary predicates]
  One of the important goals of RDF and the Semantic Web is to make (certain kinds of) statements *about* classes and predicates!

$\Rightarrow$ classes and predicates must be *in* the domain, and they must be *interpreted by the domain*

$\Rightarrow$ RDFS requires a specific model theory that is not covered by FOL.

Further reading

1. "Three Theses of Representation in the Semantic Web",
   Ian Horrocks and Peter Patel-Schneider, in *World-Wide-Web Conference (WWW 2003)*.
   Note: can be found by google or via `http://www.dblp.de`
   (discusses three ways to define a model theory for RDFS)

RDFS Axiomatic Triples

Some axioms that are expected to hold in any RDFS model can be expressed inside RDF itself independent from the chosen logic (cf. `http://www.w3.org/TR/rdf-mt`):

```
rdf:type rdfs:domain rdfs:Resource .
rdfs:domain rdfs:domain rdf:Property .
rdfs:range rdfs:domain rdf:Property .
rdfs:subPropertyOf rdfs:domain rdf:Property .
rdfs:subClassOf rdfs:domain rdfs:Class .

rdf:type rdfs:range rdfs:Class .
rdfs:domain rdfs:range rdfs:Class .
rdfs:range rdfs:range rdfs:Class .
rdfs:subPropertyOf rdfs:range rdf:Property .
rdfs:subClassOf rdfs:range rdfs:Class .

rdfs:Datatype rdfs:subClassOf rdfs:Class .
```

... and some more.

The interesting things with RDFS are not these (rather trivial) axioms, but the built-in semantics of rdfs:domain/range/subClassOf/subPropertyOf.

For them, some model theory is required.

General Considerations

Needed: a *domain* (the things talked about), and a signature (the predicate names (that are used in the *logic formalization*)).

Straightforward (and intuitive) idea

- the domain are the resources and the blank nodes,
  classes are mapped to unary predicates
  (e.g. *person(john)*), properties are mapped to binary predicates,
  (e.g. *age(john,32)* and child(john,alice)) .
- problem: rdfs:subClassOf, rdfs:range etc. talk *about* classes and properties

Alternatives (see subsequent slides)

1. FOL with reification, i.e., handling classes and properties also as domain items, and having only one meta-predicate “holds”,
2. resorting to Second-Order-Logic,
3. a special model theory not based on any other logic (as in `http://www.w3.org/TR/rdf-mt`) .

A Mapping of RDF/RDFS to FOL

- The set of constant symbols consists of all IRIs, Blank node identifiers RDF-B, and Literals RDF-L as constant symbols,
- there is a a single “holds” predicate that represents the triples.
- Herbrand-style interpretation: all terms are "interpreted by themselves", i.e., since there are no function symbols in RDF, the domain $\mathcal{D}$ *is* just the set of constant symbols (which include everything: individuals, literals, classes, properties).
- a single “holds” property that represents the triples:

$$\mathcal{I}(\text{holds}) = \{(s,p,o) | (s,p,o) \text{ holds in the given RDF ontology}\}$$

Advantages

- mapping to a well-known and well-investigated formalism,
- RDFS semantics can easily be specified by logical axioms, e.g.,
  rdfs:range specifies that the range of an rdf:Property is a certain rdfs:Class:

$$\mathcal{M} \models \forall c, p : (\ \text{holds}(p, \text{rdfs:range}, c) \rightarrow (\forall x, y : \text{holds}(x,p,y) \rightarrow \text{holds}(y, \text{rdf:type}, c)))$$

Mapping of RDF/RDFS to FOL (cont'd)

$$\mathcal{I}(\text{holds}) = \{(s,p,o)|(s,p,o) \text{ holds in the given RDF ontology}\}$$

Problems

This is just a *mapping* to an artificial predicate.

1. FOL in general is undecidable (i.e. there are no complete reasoners for it), while RDFS reasoning itself decidable, even only polynomial.
2. later: OWL builds on RDF and is based on the DL $\mathcal{SHOIN}(D)$, which is a decidable subset of FOL.
   The original translation of $\mathcal{SHOIN}(D)$ to FOL has also to be mapped to the "holds" predicate.
   Again, this would be a mapping from a decidable formalism to an undecidable one.
3. Unrestricted reification can lead to paradoxes (cf. Slide 228).
   $\Rightarrow$ Sorted FOL can be used to partition the domain into "sorts" (individuals, classes, properties) and to constrain the usage of the quantifiers.
4. Equality: for properties p_1, p_2, $p_1 = p_2$ holds by definition only if $I(p_1) = I(p_2)$ which means identity
   $\Rightarrow$ needs actually FOL+Equality

A Mapping to Higher-Order Logic

A second-order-logic domain $\mathcal{D}$ consists of two *disjoint* subsets:

- first-order objects $\mathcal{D}_1$: Let IRI$_{obj}$ and RDF-B$_{obj}$ denote all IRIs and blank nodes that denote objects. Literals also belong to that partition.
- second-order objects $\mathcal{D}_2$: predicates (and functions).
  For RDF, the predicates are classes IRI$_{cls}$ and properties IRI$_{prop}$.
  (only when defining derived classes (in OWL), there will be blank nodes RDF-B$_{cls}$ that represent classes.)
- 1st-order predicates are interpreted by relationships over the object domain.
- general: predicates of order n are interpreted over the domain of order n and are objects of the domain of order $n + 1$.
- Quantifiers range either over 1st-order objects or over 2nd-order objects, e.g.
  rdfs:range is now a 2nd-order predicate:

$$\mathcal{M} \models \forall C, P : \text{rdfs:range}(P, C) \rightarrow (\forall x, y : P(x, y) \rightarrow C(y))$$

Mapping to Higher-Order Logic (cont'd)

Assuming a given alphabet of rdfs:Classes and rdf:Properties, each of them induces a unary or binary predicate, respectively.

- Objects: $\text{IRI}_{obj} \cup \text{RDF-B}_{obj}$
- Predicates: IRI_p (note that rdf:type and rdf:Property are excluded)
- for class symbols c in IRI_c: $\mathcal{I}(c) \subseteq \text{IRI}_{obj} \cup \text{RDF-B}_{obj}$
  E.g. <foo://bla/meta#Person>(<foo://bla/persons/john>)
- for property symbols p in IRI_p ($p \neq$ rdf:Type):
  $\mathcal{I}(p) \subseteq (\text{IRI}_{obj} \cup \text{RDF-B}_{obj}) \times (\text{IRI}_{obj} \cup \text{RDF-B}_{obj} \cup \text{RDF-L})$
  E.g. <foo://bla/meta#child>(<foo://bla/persons/john>, <foo://bla/persons/alice>)
- the class rdf:Property is mapped to a 3rd-order unary predicate s.t.
  $I(\text{rdf:Property}) = \text{IRI}_{prop}$.
- rdf:type is only implicitly represented by the *interpretation* of IRI_{cls}.

Mapping to Higher-Order Logic (cont'd)

Advantages

- intuitive mapping of properties and classes
- equality: for properties p_1, p_2, $p_1 = p_2$ holds $I(p_1) = I(p_2)$ which is the case if both have the same extension,
- can also express OWL notions like transitivity of properties.

Problems

- some RDF/RDFS notions don't even fit:
  - the class rdf:Property is mapped to a 3rd-order unary predicate,
  - rdf:type would have one first-order argument and one second-order argument.
- Usage of Higher-Order Logics:
  - can be used to axiomatize complex domains, like mathematics
  - highly intractable (= non-decidable, often even no heuristics-based incomplete proof methods)
  - HOL provers exist: they are used for *interactively* proving correctness, safety etc. (e.g., HOL (1993) and Isabelle (1994))

REIFICATION

Reification (here) is applied to treat a higher-order object like a lower-order object:

- treat a class as an object, or
- treat a property as an object

i.e., to break the partitioning of the sorts/orders (which puts the mapping to Sorted FOL into FOL).

REIFICATION CAN LEAD TO PARADOXES

Reasoning with things that are both classes and instances reveals a famous paradox:

- define p as the set of all sets that do not contain themselves as an element:

  $\forall s : \ (p(s) \ \leftrightarrow \ \neg s(s))$

- is p in p?

  $p(p) \leftrightarrow \neg p(p)$

- any set of formulas that contains this definition has no model!

W3C RDF/RDFS Model Theory

RDF Semantics, W3C Recommendation 10 February 2004

(`http://www.w3.org/TR/rdf-mt`)

- a semantics and model theory for RDFS (which borrows some features from higher-order ideas) (see next slide)
- without resorting to an encoding in any other logic
  $\Rightarrow$ no possibility to use theoretical results or reasoning algorithms.

Advantages

- handles intensional equality of classes or properties,
- expresses the specific RDF/RDFS ideas

Problems

- RDF constructs are not axiomatized logically as formulas,
- but incorporated into the semantics/model theory.
- no support by any reasoner,
- not extensible/adaptable to OWL.

Aside: formal details of `http://www.w3.org/TR/rdf-mt`

An RDF/RDFS interpretation I consists of the following:

- Universe = set IR of Resources: includes resources and literals (!?)
- IS interprets URIs (= constant symbols) into the universe IR
  (`http://www.w3.org/TR/rdf-mt` defines and uses $I(X) := IS(x)$ here).
- mappings $ICEXT$ (for classes) and $IEXT$ (for properties) from IR to 2^{IR} and $2^{(IR \times IR)}$:
- the set of classes is $IC = ICEXT(IS(\text{rdfs:Class})) \subset IR$,
- for each class URI y and each URI x,
  $IS(x) \in ICEXT(IS(y)) \Leftrightarrow (IS(x), IS(y)) \in IEXT(IS(\text{rdf:type}))$
- the set of properties, $IP \subset IR$.
  for each URI x, $IS(x) \in IP \Leftrightarrow (IS(x), IS(\text{rdf:Property})) \in IEXT(IS(\text{rdf:type}))$
- for each property URI p, $IEXT(IS(p)) \subset IR \times IR$ models the triples.
- RDFS notions are not expressed by formulas, but as “semantic conditions” in the model theory, e.g.,
  - (for rdfs:range): if $(IS(x), IS(y)) \in IEXT(IS(\text{rdfs:range}))$ and
    $(IS(u), IS(v)) \in IEXT(IS(x))$ then $IS(v) \in ICEXT(IS(y))$.

5.2.2 RDFS Entailment: Practical Use of the Model Theory

An RDF Graph G *RDFS-entails* another RDF Graph H (which extends G with some edges)

- if every RDFS-interpretation which satisfies G also satisfies H.

For the *translation to FOL*, the following holds,

- if $\phi_G \cup \phi_{RDFS} \models_{FOL} \phi_H$ where ϕ_G and ϕ_H denote the FOL-translations of G and H, and ϕ_{RDFS} encodes the RDFS axioms (e.g. by rules), then $G \models_{RDFS} H$.
  [this is what rule-based reasoners do]

SPARQL on RDFS

- input: Turtle triples, seen as an RDF Graph G (including RDFS statements)
- query: a SPARQL graph pattern P
- answers: the answer bindings of all ground instances $\beta(P)$ of P s.t. $G \models_{RDFS} \beta(P)$.

$\Rightarrow$ query wrt. RDFS answering requires (a bit) more than only algebraic evaluation of conjunctive queries.

RDFS REASONING

- expressible in a decidable fragment of FOL: positive recursive Datalog
  - naive implementation: bottom-up graph completion by rules
  - querying: top-down Datalog evaluation (of any Datalog/Prolog system)
  - only issue: existentials from blank nodes (blank nodes mapped by skolemization, but it must be considered that two blank nodes describe the same individual)

$\Rightarrow$ use the FOL mapping

$\Rightarrow$ the following slides give the semantics of RDFS notions wrt. the FOL mapping.

REASONING WITH RDF, RDF SCHEMA AND OWL

- theoretical details will be discussed later. The underlying thing is either
  - graph completion by rules (RDFS, OWL-RL profile),
    (can be translated to positive Datalog)
  - *Description Logic (DL) Reasoning* (OWL DL)
    (requires a DL reasoner, based on tableaux techniques)
- there are reasoners available for the Jena Framework:
  - an internal one:
    `jena -q -inf -qf` *`sparql-file`*
    for invoking SPARQL with its internal reasoner
  - an external one:
    (integrated into the semweb.jar used in the lecture as plug-in)
    `jena -q -pellet -qf` *`sparql-file`*
    for invoking SPARQL with the Pellet DL reasoner class
  - external ones as Web Services ...

USE OF THE JENA TOOL

- option “-t”: transform (between Turtle and RDF/XML)

  jena -t -pellet -if *rdf-file* .

  (-t is not complete for checking inconsistencies)

- option “-q”: query

  jena -q -pellet [-if rdf-input-file] -qf *query-file* .

- option “-e”: export the class tree (available only when the pellet reasoner is activated).
  Input is an RDF or OWL file:

  jena -e -pellet -if *rdf-file*.

  (for checking consistency, use -e)

PELLET COMMANDLINE FOR SPARQL-DL QUERIES

- download pellet, set alias for pellet/pellet.sh
- see `pellet help` for further information
- `pellet query -q` *`query-file input-file`*
  - does not use FROM line(s) in SPARQL, input file must be given explicitly,
  - only one input file possible.

ASIDE: DIG INTERFACE - DESCRIPTION LOGIC IMPLEMENTATION GROUP

- Web page: http://dl.kr.org/dig/
- agreed "tell-and-ask-interface" of DL Reasoners as Web Service:
- tell them the facts and ask them queries, or for the whole inferred model
- e.g. supported by "Pellet"
- URL for download see Lecture Web page

```
may@dbis01:~/SemWeb-Tools/pellet-1.3$ ./pellet-dig.sh &
PelletDIGServer Version 1.3 (April 17 2006)
Port: 8081
```

- invoke the SPARQL Jena interface by
  `jena -q -qf` *`sparql-file`* `-inf -r` *`reasoner-url`*
  (e.g.: `http://localhost:8081`)
- note: the tell-functionality seems to transfer only part of the knowledge → incomplete reasoning → currently not recommended.

5.3 RDFS Vocabulary

SEMANTICS OF SUBCLASSES AND SUBPROPERTIES

rdfs:subClassOf specifies that one rdfs:Class is an rdfs:subClassOf another:

for any model $\mathcal{M}$ of the RDFS model theory,

$$\mathcal{M} \models \forall C_1, C_2 : (\ \text{holds}(C_1, \text{rdfs:subClassOf}, C_2) \rightarrow (\forall x : (\text{holds}(x, \text{rdf:type}, C_1) \rightarrow \text{holds}(x, \text{rdf:type}, C_2))))$$

rdfs:subPropertyOf specifies that one rdf:Property is an rdfs:subPropertyOf another:

$$\mathcal{M} \models \forall P_1, P_2 : (\ \text{holds}(P_1, \text{rdfs:subPropertyOf}, P_2) \rightarrow (\forall x, y : (\text{holds}(x, P_1, y) \rightarrow \text{holds}(x, P_2, y))))$$

SEMANTICS OF DOMAIN AND RANGE

rdfs:domain specifies that the domain of an rdf:Property is a certain rdfs:Class:

$$\mathcal{M} \models \forall C, P : (\ \text{holds}(P, \text{rdfs:domain}, C) \rightarrow (\forall x : (\exists y : \text{holds}(x, P, y)) \rightarrow \text{holds}(x, \text{rdf:type}, C)))$$

rdfs:range specifies that the range of an rdf:Property is a certain rdfs:Class
(note that rdfs:Datatype is a subclass (and an instance) of rdfs:Class):

$$\mathcal{M} \models \forall C, P : (\ \text{holds}(P, \text{rdfs:range}, C) \rightarrow (\forall y : (\exists x : \text{holds}(x, P, y)) \rightarrow \text{holds}(y, \text{rdf:type}, C)))$$

Exercise

- Give an implementation by Datalog Rules for RDFS constructs.

SUBCLASS, DOMAIN, RANGE: EXAMPLE

```
@prefix : <foo://bla/meta#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
  :has_cat rdfs:domain :Person .
  :has_cat rdfs:range :Cat .
  :Person rdfs:subClassOf :LivingBeing .
  :Cat rdfs:subClassOf :LivingBeing .
  <foo://bla/persons/john> :has_cat <foo://bla/cats/garfield>.
  <foo://bla/persons/mary> rdf:type :Person.
```

[Filename: RDF/subclass.n3]

```
prefix : <foo://bla/meta#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X ?T
from <file:subclass.n3>
where {?X rdf:type ?T}
```

[Filename: RDF/subclass.sparql]

- activate the reasoner (internal or pellet) when invoking Jena.

SUBCLASS, DOMAIN, RANGE: EXAMPLE (CONT'D)

Recall the previous example. Given the following facts:

```
:has_cat rdfs:domain :Person .
:has_cat rdfs:range :Cat .
:Person rdfs:subClassOf :LivingBeing .
:Cat rdfs:subClassOf :LivingBeing .
<foo://bla/persons/john> :has_cat <foo://bla/cats/garfield>.
<foo://bla/persons/mary> rdf:type :Person.
```

The domain/range information does not act as a constraint, but as information. From that knowledge, the following facts can be *inferred*:

- :has_cat implies that the subject (John) is a Person, and the object (Garfield) is a cat,
- both are thus LivingBeings.

SUBPROPERTIES

- outlook: combine it with owl:TransitiveProperty.

```
@prefix : <foo://bla/meta#> .
@prefix p: <foo://bla/persons/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
  p:john :hasChild p:alice, p:bob.
  p:kate :hasChild p:john.
  :hasChild rdfs:subPropertyOf :descendant.
  :descendant rdf:type owl:TransitiveProperty.
```

[Filename: RDF/descendants.n3]

```
prefix : <foo://bla/meta#>
select ?X ?Y
from <file:descendants.n3>
where {?X :descendant ?Y}
```

[Filename: RDF/descendants.sparql]

COMPARISON

SQL

- queries only against the database (no intensional knowledge),
- equivalent to tree expressions in relational algebra, based on set theory,
- formal semantics can be given purely syntactically with the algebra,

$\Rightarrow$ in the DB lecture, we did not need logic.

- equivalent to the relational calculus, semantics of queries can be given by the calculus. Equivalent to *nonrecursive Datalog* (cf. Database Theory Lecture) with “negation as failure” (top-down) stratification (bottom-up).

RDFS + SPARQL

- only restricted negation
- RDFS: built-in rules (positive, recursive Datalog)
- SPARQL: positive, nonrecursive Datalog
- intuitive bottom-up semantics

USING RDF IN THE WORLD WIDE WEB

- The (Semantic) Web is not seen as a collection of documents, but as a collection of correlated information (described via documents)
- using RDF, everybody can make statements about any resource
  (cf. link-bases in XLink)
  - incremental, world wide data and meta-data
  - distributed RDFS,
  - distributed RDF,
  - real URLs (Linked Open Data; cf. Slides 286 ff) vs. only virtual resources (URIs).
- not assumed that complete information about any resource is available.
- Open world, no notion of (implicit) negation.
- potentially inconsistent information;
- statements can be equipped with probabilities or labeled as opinions;
  fuzzy reasoning, belief revision ...
- ... lots of artificial intelligence applications ...

REASONING BASED ON RDFS

- RDF/RDFS *model theory* as above,
- rather simple Datalog rules, graph completion,
- queries: against the (completed) graph by matching (SPARQL).
- incomplete knowledge when reasoning: “open world assumption”
- only very restricted negation (none in RDFS; “!bound” in SPARQL 1.0 and “FILTER NOT EXISTS” in SPARQL 1.1 allow for negation of failure)

Preview: OWL

- based on DLs
- OWL-DL includes constructs that are not expressible by Datalog (union/disjunction) – requires “Disjunctive Datalog”
  $\Rightarrow$ totally different complexity and reasoning algorithms
- OWL-Lite is a simpler fragment (e.g., only 0-1-*-cardinalities) ... turned out that tools are not much simpler to design
- The OWL-RL profile: can be translated to positive recursive Datalog.

EXAMPLE/EXERCISE

Consider again the employee-manages-departments example (Slide 22).

- Give the RDF Graph.
- give the Turtle triples and feed them into the Jena tool.

ADDITIONAL RDF/RDFS VOCABULARY

The rdf/rdfs namespaces provide some more vocabulary:

Like most data models, RDF provides a representation for *Collections*:

- Collections: `rdf:Alt`, `rdf:Bag`, `rdf:Seq`, `rdf:List` are collections.
  Lists have properties `rdf:first` (a resource) and `rdf:rest` (a list). Others have properties `_1`, `_2`, ... that refer to their members.
- (rdfs:Container, rdfs:member, rdfs:ContainerMembershipProperty)

... these are partially used implicitly (e.g., collections in owl:intersectionOf, owl:OneOf), but often not supported by OWL reasoners if used explicitly (see Slides 430 ff.).

EXAMPLE: THE MONDIAL ONTOLOGY

See mondial.n3, mondial-europe.n3 and mondial-meta.n3 on the Web page.

Note that it is highly redundant: defining just rdfs:domain and rdfs:range of properties implies most of the classes (and also most of the rdfs:type relationships in mondial.n3).

```
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X
from <file:mondial.n3>
from <file:mondial-meta.n3>
where {?X rdf:type mon:Country}
```

[Filename: RDF/mondial-meta-query.sparql]

- activate Jena with reasoner (if mondial.n3 is too big, use mondial-europe.n3 instead)

Mondial is not an interesting example for RDFS (and OWL):

- it's mainly data, no intensional knowledge, no complex ontology
- for that reason it is a good example for SQL and XML.
- RDFS and OWL is interesting when information is *combined* and additional knowledge can be derived.

Developing Ontologies

- have an idea of the required concepts and relationships (ER, UML, ...),
- generate a (draft) n3 or RDF/XML instance,
- write a separate file for the metadata,
- load it into Jena with activating a reasoner.
- If the reasoner complains about an inconsistent ontology, check the metadata file alone. If this is consistent, and it complains only when also data is loaded:
  - it may be due to populating a class whose definition is inconsistent and that thus must be empty.
  - often it is due to wrong datatypes. Recall that datatype specification is not interpreted as a constraint (that is violated for a given value), but as additional knowledge.

Chapter 6
RDF/XML: XML Syntax of RDF Data

- An XML representation of RDF data for providing RDF data on the Web

$\Rightarrow$ could be done straightforwardly as a “holds” relation mapped according to SQLX (see next slide).

- would be highly redundant and very different from an XML representation of the same data
- search for a more similar way: leads to “striped RDF/XML”
  - data feels like XML: can be queried by XPath/Query and transformed by XSLT
  - can be parsed into an RDF graph.
- usually: provide RDF/XML data to an agreed RDFS/OWL ontology.

A Straightforward XML Representation of RDF Data

Note: this is not RDF/XML, but just some possible representation.

- RDF data are triples,
- their components are either URIs or literals (of XML Schema datatypes),
- straightforward XML markup in SQLX style,
- since Turtle has a term structure, it is easy to find an XML markup.

```
<my-n3:rdf-graph xmlns:my-n3="http://simple-silly-rdf-xml.de#">
 <my-n3:triple>
  <my-n3:subject type="uri">foo://bar/persons/john</my-n3:subject>
  <my-n3:predicate type="uri">foo://bar/meta#name</my-n3:predicate>
  <my-n3:object type="http://www.w3.org/2001/XMLSchema#string">John</my-n3:object>
 </my-n3 triple>
 <my-n3:triple> ... </my-n3 triple>
   :
</my-n3:rdf-graph>
```

- The problem is not to have *any* XML markup, but to have a useful one that covers the *semantics* of the RDF data model.

6.1 RDF/XML: RDF as an XML Application

- root element type: <rdf:RDF>
  (namespace URL: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
- not just “some markup”
- but covers the semantics of “resource description”

Markup

- “Striped RDF/XML” syntax as an abbreviated form (similar to the well-known XML structure)

RDF/XML Descriptions of Resources

<rdf:Description> elements collect a (partial) description of a *resource*:

- which resource is described: @rdf:about="*uri*"
- subelements describe its properties (amongst them, its type as a special property),
  - element name: name of the property
    Note that this name is actually an URI.
    (this is where XML namespaces come into play)
  - value of the property:
    * element contents:
      text content or one or more nested <rdf:Description> elements
    * attribute @rdf:resource="*uri*": property points to another resource that has an RDF description of its own elsewhere
- can contain nested <rdf:Description> elements similar to blank nodes in nested Turtle data.
- there can be multiple descriptions of the same resource (as in Turtle).
- later: different URI definition mechanisms

Example

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xml:base="foo://bla/"
         xmlns="foo://bla/meta#">
 <rdf:Description rdf:about="persons/john">
  <rdf:type rdf:resource="meta#Person"/>
  <name>John</name>
  <age>35</age>
  <hasChild>
   <rdf:Description rdf:about="persons/alice">
    <rdf:type rdf:resource="meta#Person"/>
    <name>Alice</name>
    <age>10</age>
   </rdf:Description>
  </hasCild>
  <hasChild rdf:resource="persons/bob"/>
 </rdf:Description>
 <rdf:Description rdf:about="persons/bob">
  <rdf:type rdf:resource="meta#Person"/>
  <name>Bob</name>
  <age>8</age>
 </rdf:Description>
</rdf:RDF>
```

[Filename: RDF/john-rdfxml.rdf]

- xml:base determines the URI prefix, either flat (ending with a “#”, or hierarchical, ending with a “/”)
- in 2nd case: local parts can be hierarchical expressions
- default namespace set to <foo://bla/meta#>
- element names are the property names

```
prefix : <foo://bla/meta#>
select ?X ?Y ?A
from <file:john-rdfxml.rdf>
where {?X :hasChild ?Y . ?Y :age ?A}
```

[Filename: RDF/john-rdfxml.sparql]

ABBREVIATED FORM: STRIPED RDF/XML

- Full syntax:

  <rdf:Description rdf:about="*uri*">
  <rdf:type rdf:resource="*classname*">
  resource description
  </rdf:Description>

- Abbreviated syntax:

  <*classname* rdf:about="*uri*">
  resource description
  </*classname*>

- Striped RDF/XML: alternatingly *classname* – *propertyname* – *classname*
- domain terminology URIs = element names
- all attribute names are in the RDF namespace
- all object URIs are in attribute values
- all attribute values are object URIs
  (next: an even shorter form where this will not hold!)

Example: Striped

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xml:base="foo://bla/persons/"
   xmlns="foo://bla/meta#">
 <Person rdf:about="john">
  <name>John</name>
  <age>35</age>
  <hasChild>
   <Person rdf:about="alice">
    <name>Alice</name>
    <age>10</age>
   </Person>
  </hasChild>
  <hasChild rdf:resource="bob"/>
 </Person>
 <Person rdf:about="bob">
  <name>Bob</name>
  <age>8</age>
 </Person>
</rdf:RDF>
```

[Filename: RDF/john-striped.rdf]

- looks very much like well-known XML
- xml:base applies now only to objects' URIs
  e.g. <foo://bla/persons/alice>
- terminology URIs reside all in the namespaces
- same query as before:

```
# jena -q -qf john-striped.sparql
prefix : <foo://bla/meta#>
select ?X ?Y
from <file:john-striped.rdf>
where {?X :hasChild ?Y}
```

[Filename: RDF/john-striped.sparql]

ABBREVIATED FORM: STRIPED RDF/XML WITH VALUE ATTRIBUTES

- Full syntax:

  <rdf:Description rdf:about="*uri*">
  <rdf:type rdf:resource="*classname*"
  <property$_1$>*value*</property$_1$>
  <property$_2$ rdf:resource="*uri*"/>
  </rdf:Description>

  where property$_1$ has a single, scalar value (string or number)

- Abbreviated syntax:

  <*classname* rdf:about="*uri*" prefix:property$_1$="*value*">
  <property$_2$ rdf:resource="*uri*"/>
  </*classname*>

- Striped RDF/XML: alternatingly *classname* – *propertyname* – *classname*
- domain terminology URIs = element and attribute names
  Note: attributes MUST be prefixed by an explicit namespace
- attribute values are object URIs or literal values.

Example: Striped with Attributes

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xml:base="foo://bla/persons/"
   xmlns:p="foo://bla/meta#">
 <p:Person rdf:about="john" p:name="John" p:age="35">
  <p:hasChild>
   <p:Person rdf:about="alice" p:name="Alice" p:age="10"/>
  </p:hasChild>
  <p:hasChild rdf:resource="bob"/>
 </p:Person>
 <p:Person rdf:about="bob" p:name="Bob" p:age="8"/>
</rdf:RDF>
```

[Filename: RDF/john-striped-attrs.rdf]

- looks even more like well-known XML

```
# jena -q -qf john-striped-attrs.sparql
prefix : <foo://bla/meta#>
select ?X ?Y ?N
from <file:john-striped-attrs.rdf>
where {?X :hasChild ?Y . ?Y :name ?N}
```

[Filename: RDF/john-striped-attrs.sparql]

ABBREVIATIONS

- omit “blank” description nodes by
  <*property-name* rdf:parseType=“Resource”> ... </*property-name*>
- literal-valued properties can even be added to the surrounding property element.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:mon="http://www.semwebtech.org/mondial/10/meta#">
 <mon:City rdf:nodeID="hannover" mon:name="Hannover">
  <mon:population rdf:parseType="Resource">
   <mon:year>1995</mon:year> <mon:value>525763</mon:value>
  </mon:population>
  <mon:population mon:year="2002" mon:value="515001"/>
 </mon:City>
</rdf:RDF>
```

[Filename: RDF/parse-type.rdf]

- rdf:parseType is not a real RDF citizen:
  - it exists only in the RDF/XML serialization,
  - it is intended as a parsing instruction to the RDF/XML $\rightarrow$ RDF parser.

URI Representation/Construction Mechanisms

- describe a remote resource via its full global URI (as above)
  - attribute @rdf:about="*uri*" identifies a remote resource
- use a base URI by xml:base that sets the base URI for resolving relative RDF URI references (i.e., rdf:about, rdf:resource, rdf:ID and rdf:datatype), otherwise the base URI is that of the document.
  - set xml:base="*uri*" (e.g. in the root element)
  - @rdf:about="*relativepath*": the resource's global URI is then composed as *xmlbase relativepath* (note that *xmlbase* must end with "/" or "#")
  - @rdf:ID="*local-id*": the resource's global URI is then composed as *xmlbase#local-id*. *local-id* must be a simple qname (no path!)
  - then, use @rdf:resource="*#localpart*" in the object position for referencing it.
- only locally known IDs:
  - attribute @rdf:nodeID= "*name*": defines and describes a *local* resource that can be referenced only inside the same RDF instance by its ID
  - then, use @rdf:nodeID="*id*" in the object position of a property instead of @rdf:resource="*uri*"

Example: using global protocol://path#IDs

- does only work with #-namespaces, otherwise constructs foo://bla/persons/#john

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xml:base="foo://bla/flatpersons#"
   xmlns="foo://bla/meta#">
 <Person rdf:ID="john">
  <name>John</name>
  <age>35</age>
  <hasChild>
   <Person rdf:ID="alice">
    <name>Alice</name>
    <age>10</age>
   </Person>
  </hasChild>
  <hasChild rdf:resource="#bob"/>
 </Person>
 <Person rdf:ID="bob">
  <name>Bob</name>
  <age>8</age>
 </Person>
</rdf:RDF>
```

[Filename: RDF/john-ids.rdf]

- xml:base = “foo://bla/flatpersons#” determines the URI prefix.
  IDs must then be qnames (e.g. “john/doe” not allowed)
- default namespace set to <foo://bla/meta#>
- element names are the property names

```
# jena -q -qf john-ids-rdf.sparql
prefix : <foo://bla/meta#>
select ?X ?Y
from <file:john-ids.rdf>
where {?X :hasChild ?Y}
```

[Filename: RDF/john-ids-rdf.sparql]

- URIs are then <foo://bla/flatpersons#john>and <foo://bla/meta#name>

Example: using local IDs

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns="foo://bla/meta#">
 <Person rdf:nodeID="john">
  <name>John</name>
  <age>35</age>
  <hasChild>
   <Person rdf:nodeID="alice">
    <name>Alice</name>
    <age>10</age>
   </Person>
  </hasChild>
  <hasChild rdf:nodeID="bob"/>
 </Person>
 <Person rdf:nodeID="bob">
  <name>Bob</name>
  <age>8</age>
 </Person>
</rdf:RDF>
```

[Filename: RDF/john-local.rdf]

- no xml:base
- all IDs must be qnames and are localized (e.g., _:b1)
- default namespace set to “foo://bla/meta#”
- element names are the property names

```
# jena -q -qf john-local-rdf.sparql
prefix : <foo://bla/meta#>
select ?X ?Y ?N
from <file:john-local.rdf>
where {?X :hasChild ?Y. ?Y :name ?N}
```

[Filename: RDF/john-local-rdf.sparql]

- a result of the query is e.g. ?X/_:b0, ?Y/_:b1, ?N/“Bob”
- these local resources cannot be referenced by other RDF instances.

Example (with base URI and relative paths)

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:mon="http://www.semwebtech.org/mondial/10/meta#"
   xml:base="http://www.semwebtech.org/mondial/10/">
 <mon:Country rdf:about="countries/D/" mon:name="Germany" mon:code="D">
  <mon:hasProvince>
   <mon:Province rdf:about="countries/D/provinces/Niedersachsen/" mon:name="Niedersachsen">
    <mon:hasCity>
     <mon:City rdf:about="countries/D/provinces/Niedersachsen/cities/Hannover/" mon:name="Hannover">
      <mon:population>
       <rdf:Description>
        <mon:year>1995</mon:year> <mon:value>525763</mon:value>
       </rdf:Description>
      </mon:population>
     </mon:City>
    </mon:hasCity>
    <mon:capital rdf:resource="countries/D/provinces/Niedersachsen/cities/Hannover/"/>
   </mon:Province>
  </mon:hasProvince>
 </mon:Country>
</rdf:RDF>
```

[Filename: RDF/a-bit-mondial.rdf]

- global URIs are e.g. <http://www.semwebtech.org/mondial/10/meta#name>and <http://www.semwebtech.org/mondial/10/countries/D/provinces/Niedersachsen/cities/Hannover>
- rdf:Description used for a blank node (population) – this will even be shorter later

NAMES VS. URIS – XMLNS VS. XML:BASE

- element and attribute names are subject to namespace expansion,
- URIs in rdf:about, rdf:resource, rdf:ID and rdf:datatype are subject to expansion with xml:base.

- What if URIs from different areas are used?
  - inside a document, different (even hierarchically nested!) xml:base values can be used,
  - entities can be used inside URIs.

LOCAL XML:BASE VALUES

- here, it pays that with the XML level, there is an intermediate semantical level (in contrast to the pure Turtle syntax)

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:mon="http://www.semwebtech.org/mondial/10/meta#"
   xml:base="http://www.semwebtech.org/mondial/10/">
 <mon:Country xml:base="countries/D/" rdf:about="." mon:name="Germany" mon:code="D">
  <mon:has_city>
   <mon:City rdf:about="cities/Berlin" mon:name="Berlin"/>
  </mon:has_city>
 </mon:Country>
 <mon:Country xml:base="foo://bla/countries/F/" rdf:about="." mon:name="France" mon:code="F">
  <mon:has_city>
   <mon:City rdf:about="cities/Paris" mon:name="Paris"/>
  </mon:has_city>
 </mon:Country>
</rdf:RDF>
```

[Filename: RDF/url-expansion.rdf]

- relative xml:base expressions are appended: <http://www.semwebtech.org/mondial/10/countries/D/cities/Berlin>
- absolute xml:base expressions overwrite: <foo://bla/countries/F/cities/Paris>.

XML Entities in URIs

- if URIs from different bases are mingled in the document:

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY mon "http://www.semwebtech.org/mondial/10/">
  <!ENTITY xyz "a:bc"> ] >
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:mon="this://is-actually-not-used"
         xmlns:f="foo://bla#"
         xml:base="foo://bla/">
 <f:Person rdf:about="persons/john" f:name="John" f:age="35">
  <!-- this is not expanded at all: -->
  <f:test rdf:resource="mon:countries/D/cities/Berlin"/>
  <!-- the right way is to use an entity: -->
  <f:lives-in rdf:resource="&mon;countries/D/cities/Berlin"/>
  <f:married-to rdf:resource="&xyz;#mary"/>
 </f:Person>
</rdf:RDF>
```

[Filename: RDF/url-entities.rdf]

```
# jena -q -qf url-entities.sparql
select ?X ?P ?Y
from <file:url-entities.rdf>
where {?X ?P ?Y}
```

[Filename: RDF/url-entities.sparql]

Specification of Datatypes in RDF/XML

- RDF uses XML Schema types
- yields typed literals such as “42”^^<http://www.w3.org/2001/XMLSchema#int>
- In RDF/XML, the type of a literal value is specified by an rdf:datatype attribute whose value is recommended to be one of the following: an XSD literal type URI or the URI of the datatype rdf:XMLLiteral.
  (but then, they cannot be abbreviated into attributes)

```
<mon:Country rdf:resource=“http://www.semwebtech.org/mondial/10/countries/D”>
  <mon:name
    rdf:datatype=“http://www.w3.org/2001/XMLSchema#string”>Germany</mon:name>
  <mon:area
    rdf:datatype=“http://www.w3.org/2001/XMLSchema#float”>356910</mon:area>
</mon:Country>
```

[example next slide]

DATATYPES: EXAMPLE

note: `http://www.w3.org/2001/XMLSchema#` can be defined as an entity in the local DTD to the RDF/RDFS instance and is then used as rdf:datatype="&xsd;string"

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#"> ]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:mon="http://www.semwebtech.org/mondial/10/meta#"
   xml:base="http://www.semwebtech.org/mondial/10/">
 <mon:Country rdf:about="countries/D">
  <mon:name rdf:datatype="&xsd;string">Germany</mon:name>
  <mon:population rdf:datatype="&xsd;int">83536115</mon:population>
 </mon:Country>
</rdf:RDF>
```

[Filename: RDF/rdf-datatype.rdf]

- `jena -t -pellet -if rdf-datatype.rdf`
- Note: having linebreaks in the data yields unexpected results.

XMLLITERAL IN RDF/XML: EXAMPLE

- use rdf:parseType="Literal":

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xml:base="foo://bla/persons/"
   xmlns:p="foo://bla/meta#">
 <p:Person rdf:about="john" p:name="John" p:age="35">
  <p:homepage rdf:parseType="Literal">
   <ht:html xmlns:ht="http://www.w3.org/1999/xhtml">
    <ht:body><ht:li>bla</ht:li></ht:body>
   </ht:html>
  </p:homepage>
  <p:hasChild rdf:resource="alice"/>
 </p:Person>
</rdf:RDF>
```

[Filename: RDF/rdf-xmlliteral.rdf]

```
prefix : <foo://bla/persons/>
select ?X ?P ?Y
from <file:rdf-xmlliteral.rdf>
where {:john ?P ?Y}
```

[Filename: RDF/rdf-xmlliteral.sparql]

- the resulting literal is
  "<ht:html . . . > . . . </ht:html>"^^<http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral>
- ... including the newlines (= XML text nodes) inside the XML fragment.

Using XMLLiterals: Exclusive Canonical XML

- Required by some software (e.g., in the “Jena” Semantic Web Framework)
- XML fragments/subtrees must be processable without their context – thus, namespaces must be present at appropriate levels in the tree.
- Details: `http://www.w3.org/TR/xml-exc-c14n/`
- can be obtained with `xmllint -exc-c14n x.xml > y.xml` (and analogously by other tools)

RDF/XML VS. "PURE" XML

- striped RDF/XML gives very much the look&feel of common XML documents:
  - nearly no "rdf:..." elements
  - no "rdf:..." elements that are relevant from the XML processing point of view
- can be processed with XPath/XQuery and XSLT as pure XML data
- can also be processed as RDF data in *combination* with RDFS/OWL metadata information (usually from a different source).

MACHINE-READIBILITY

- RDF/XML is usually automatically generated,
- not intended to be read by humans,
- processes as XML serialization of RDF data
  - as a file: $\rightarrow$ RDF/XML parser $\rightarrow$ RDF graph
  - as an (XML!) stream: possible preprocessing and then mapping to a graph/DB
  - use generic tools for XML stream processing, like SAX/StAX or the XML Digester (see slides for XML lecture/XML lab course)
    - generic processing stream$\rightarrow$RDF graph
    - preprocessing stream$\rightarrow$ filtered/modified graph
      (but note that nodes can occur in arbitrary order)

    Practical Exercise: Write an XML/RDF Parser in SAX/StAX/Digester.

6.2 XML Syntax of RDFS/OWL

- RDFS/OWL descriptions are also <rdf:Description>s – descriptions of types/rdfs:/owl:Classes or rdf:Properties
- additionally include rdfs namespace declaration

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
```

same as above:

- Full syntax:

```
<rdf:Description rdf:about="class-uri">
  <rdf:type rdf:resource="owl:Class">
    resource description
</rdf:Description>
```

- Abbreviated syntax:

```
<owl:Class rdf:about="class-uri">
    resource description
</owl:Class>
```

- Full syntax:

```
<rdf:Description rdf:about="property-uri">
  <rdf:type rdf:resource="rdf:Property">
    resource description
</rdf:Description>
```

- Abbreviated syntax:

```
<rdf:Property rdf:about="property-uri">
    resource description
</rdf:Property>
```

RDF Schema Documents

- description of classes

  <owl:Class rdf:about="*uri*">
  <rdfs:subClassOf rdf:resource="*class-uri*$_2$"/>
  </owl:Class>

  used in RDF/XML data documents by <rdf:type resource="*uri*"/> or <*uri*>...</*uri*>, also used by <rdfs:subClassOf rdf:resource="*uri*"/> (and by rdfs:domain and rdfs:range).

- description of properties

  <rdf:Property rdf:about="*uri*">
  <rdfs:subPropertyOf rdf:resource="*property-uri*$_2$"/>
  <rdfs:domain rdf:resource="*class-uri*$_1$"/>
  <rdfs:range rdf:resource="*class-uri*$_2$"/>
  </rdf:Property>

  used by names of property elements and of property attributes in RDF/XML data documents, and for <rdfs:subPropertyOf rdf:resource="*uri*"/>.

- instead of @rdf:about="*uri*" the notations xml:base + local part or local-ids can be used.

- further subelements for class and property descriptions are provided by OWL.

DEFINING URIS OF RDFS CLASSES AND PROPERTIES

Classes and properties are “usual” resources, identified/defined by

<owl:Class rdf:about=“*class-uri*”> . . . </owl:Class>
(absolute URI or relative URI + base-URI)
reference by rdf:resource=“*class-uri*”

<owl:Class rdf:ID=“*classname*”> . . . </owl:Class> (+ base-URI or extending the file’s URL)
reference by rdf:resource=“*#classname*” (local)
reference by rdf:resource=“*base-uri#classname*” (from remote)

<owl:Class rdf:nodeID=“*classname*”> . . . </owl:Class>
reference by rdf:nodeID=“*classname*” (only for local definitions)

(analogous for <rdf:Property>)

Version A: Classes and Properties URLs refer to the same file

- use rdf:ID and rdf:resource="#..."

Anything that is defined in an RDFS/OWL document - e.g., in

<http://foo/bla/metatest.rdf> (or with appropriate setting of xml:base)

```
<owl:Class rdf:ID=" Country "> <!-- subClassOf-defs etc.--> </owl:Class>
<rdf:Property rdf:ID=" capital ">
    <rdfs:domain rdf:resource="#Country"/>
    <rdfs:range rdf:resource="#City"/>
</rdf:Property>
```

defines URIs <http://foo/bla/metatest.rdf#Country> and <http://foo/bla/metatest.rdf#capital> etc. that can be used in another RDF document as (the same applies to the Turtle format)

```
<rdf:RDF xmlns:m="http://foo/bla/metatest.rdf#"
      xml:base="http://www.semwebtech.org/mondial/10/>
  < mon:Country rdf:about="countries/D" mon:name = "Germany">
    < mon:capital rdf:resource="countries/D/provinces/Berlin/cities/Berlin"/>
  </mon:Country>
</rdf:RDF>
```

Example

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#">
 <owl:Class rdf:ID="Country"/>
 <owl:Class rdf:ID="City"/>
 <rdf:Property rdf:ID="capital">
   <rdfs:domain rdf:resource="#Country"/>
   <rdfs:range rdf:resource="#City"/>
 </rdf:Property>
</rdf:RDF>
```

[Filename: RDF/metatest.rdf]

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:m="file:///home/may/teaching/SemWeb/RDF/metatest.rdf#"
   xml:base="bla:dontmatterwhat#">
 <m:Country rdf:ID="Germany">     <!-- bla:dontmatterwhat#Germany -->
   <m:capital rdf:nodeID="Berlin"/>   <!-- local blank node _:b0 -->
 </m:Country>
</rdf:RDF>
```

[Filename: RDF/metatest2.rdf]

Example (cont'd)

```
select ?X ?P ?Y
from <file:metatest.rdf>
from <file:metatest2.rdf>
where {?X ?P ?Y}
```

[Filename: RDF/metatest.sparql]

- call `jena -q -qf metatest.sparql`
- illustrates the constructed URLs, like
  `<file:///home/may/teaching/SemWeb/RDF/metatest.rdf#capital>`
  (using xml:base or an http: URL makes more sense in reality)
- call `jena -pellet -q -qf metatest.sparql`
- lists lots of "derived" OWL axioms, but also derives that
  `_:b0 a <file:///home/may/teaching/SemWeb/RDF/metatest.rdf#City>`

Version B: Virtual (URI) or remote (URL) Resources

Using rdf:about in a class definition specifies anything about a remote resource URL/URI:

- straightforward by <owl:Class rdf:about="*class-uri*"> and <owl:Property rdf:about="*property-uri*">
- write the complete URI, or
- use appropriate xml:base or entities (RDF/XML), or base and prefixes (Turtle).

Comparison

- Version A: class/property resources are represented by XML fragments of the RDFS+OWL instance:
  - when directly extending the file's URL:
    @rdf:resource similar to an HTML anchor inside the XML document, contains the class/property definition
    (This matches the original HTML+XML flavor of the late 1990s, but such HTML-style derefencing is not implemented by default, Linked Open Data turned out to work in a different way based on Web Services)
  - only practical if the RDFS+OWL is non-distributed
    (although remote RDFS+OWL instances can also add to this resource's description by using rdfs:about)

  $\Rightarrow$ centralized ontologies
- Version B: class/property resources are identified by a virtual URI
  - arbitrary RDFS instances can contribute to the resource description
  - users/clients have to know where the resource descriptions can be found

  $\Rightarrow$ distributed ontologies (only a common syntax pattern for class/property/instance URIs is required)

USE CASES FOR CLASS URIS IN RDF/XML DOCUMENTS

- in striped RDF/XML instance data documents by class elements:
  <*[nsprefix:]classname*> . . . </*[nsprefix:]classname*>
  - non-prefixed: *default-namespace+classname* yields the *class-uri*.
  - prefixed: *nsprefix-namespace+classname* yields the *class-uri*.
- in striped RDF/XML instance data documents by <rdf:type rdf:resource="*class-uri*"/>
  - expanded wrt. xml:base; but usually the xml:base of the instance data document is different from the base of the domain names (=namespace). Use an entity if needed.
- references from RDFS+OWL documents by
  <rdfs:subClassOf rdf:resource="*class-uri*"/>
  (analogously for rdfs:domain and rdfs:range and OWL properties)
  - expanded wrt. xml:base.
- incremental RDFS/OWL descriptions of the same class in RDFS/OWL documents by
  <rdf:Description rdf:about="*class-uri*">. . . </rdf:Description>
  - expanded wrt. xml:base.
- (note that in Turtle, the user can decide to expand wrt. <relURL>+@base or prefix:qname)

USE CASES FOR PROPERTY URIS IN RDF/XML DOCUMENTS

- in striped RDF/XML instance data documents by property subelements or attributes:

  <*surrounding-element [nsprefix:]propertyname*="...">
  <*[nsprefix:]propertyname*> . . . </*[nsprefix:]propertyname*>
  :
  </*surrounding-element*>

  - non-prefixed: *default-namespace+classname* yields the *class-uri*.
  - prefixed: *nsprefix-namespace+elementname* yields the *property-uri*.

- references from RDFS+OWL RDF/XML documents by
  <rdfs:subPropertyOf rdf:resource="*property-uri*"/>

  - expanded wrt. xml:base.

- incremental RDFS+OWL descriptions of the same property in RDF/XML documents by
  <rdf:Description rdf:about="*class-uri*">. . . </rdf:Description>

  - expanded wrt. xml:base.

USE CASES FOR XML SCHEMA DATATYPES IN METADATA

- For literal properties, the domain of <rdf:Property> can refer to XML Schema types, e.g.

```
<rdf:Property rdf:ID="indepDate">
    <rdfs:domain rdf:resource="#Country"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#date"/>
</rdf:Property>
```

6.3 Example Sketch: World Wide RDF Web

- many information sources that describe resources
- higher level information management (e.g., portals): use some of these sources for accessing *integrated* information

Example (RDF source see next slide) – the example is not based on real data

- mondial: countries, cities
- `<http://www.semwebtech.org/mondial/10/meta#>`: the geography ontology
- another resource: cities and their airports
- `<http://sw.iata.org/ontology#>` (International Air Transport Assoc.): ontology about flight information
- `<bla://sw.iata.org/flights/>`*flight*: resource associated with a given flight (e.g. LH42).
- `<bla://sw.iata.org/airports/>`*abbrev*: resource associated with a given airport (e.g., FRA, CDG).
- there will probably be a Lufthansa RDF database that describes the flights in their terminology

Example (Cont'd) [Filename: RDF/flightbase.rdf]

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
  <!ENTITY mon "http://www.semwebtech.org/mondial/10/"> ]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:mon="http://www.semwebtech.org/mondial/10/meta#"
   xmlns:travel="http://www.semwebtech.org/domains/2006/travel#">
 <rdf:Description rdf:about="&mon;countries/D/provinces/Berlin/cities/Berlin">
   <travel:has_airport rdf:resource="bla://sw.iata.org/airports/BLN"/>
 </rdf:Description>
 <rdf:Description rdf:about="&mon;countries/F/provinces/IledeFrance/cities/Paris">
  <travel:has_airport rdf:resource="bla://sw.iata.org/airports/CDG"/>
 </rdf:Description>
 <rdf:Description rdf:about="bla://sw.iata.org/flights/LH42"
      xmlns:iata="http://sw.iata.org/ontology#">
  <rdf:type rdf:resource="http://sw.iata.org/ontology#Flight"/>
  <iata:from rdf:resource="bla://sw.iata.org/airports/BLN"/>
  <iata:to rdf:resource="bla://sw.iata.org/airports/CDG"/>
 </rdf:Description>
</rdf:RDF>
```

RDF vs. XML

Everything that can be expressed by XML can also be expressed by RDF

+ RDF can also be used to *describe* resources
  (pictures, movies, ..., programs, Web services, ...)
+ RDF can be represented as a graph, independent from the structure of the (distributed) RDF instances
+ RDF data can be distributed over different files that describe the same resources
+ RDF has a connection to global schema description mechanisms
o RDF/XML can be queried in the same way by XPath/XQuery ...
- but: which RDF and RDFS/OWL instances?
  – if local resources are used: relatively easy
  – if global resources are used: appropriate RDFs must be searched for.

Chapter 7
Linked Open Data – World Wide RDF Web – Web of Data

Traditional RDF Data Sources

- Web sources (Web services or “traditional” Web Servers) provide HTML (for browsers) or RDF (for data processing) data.
- simple, via physical resources of the form `filename.html`, `filename.rdf`, or `filename.ttl` etc.
- virtual URIs (and also URLs) are used inside of the the data.

LINKED OPEN DATA (LOD) PRINCIPLES

Mandatory: access to RDF data

- There must be resolvable http:// (or https://) URIs.
- They must resolve, with or without content negotiation, to RDF data in one of the popular RDF formats (RDFa, RDF/XML, Turtle, N-Triples).
- Access of the entire dataset must be possible via RDF crawling, via an RDF dump, or via a SPARQL endpoint.
- There should be references (URL links) to other LOD datasets
  - properties whose object is an URL from another dataset, or
  - owl:sameAs triples that connect to URL from another dataset

Optional: access as HTML Web pages based on the RDF data

- pages about individual resources
- SPARQL Web interface
- References:
  `https://www.w3.org/DesignIssues/LinkedData.html`
  `http://www.lod-cloud.net/#about`

LOD: Access Options

- Example case: Mondial as LOD

RDF Data Access

- The URIs in Mondial are "real" URLs of the form
  `http://www.semwebtech.org/mondial/10/countries/D/` for Germany
- URL access with a (HTML) Web browser: Web page with formatted information about this object.
- URL access with an RDF client: receives RDF data related to this object (commonly sent as XML/RDF)
  - "rapper" tool: `rapper http://www.semwebtech.org/mondial/10/islands/Sicilia/` (shows the data in Turtle format)
  -
```
# jena -q -if http://www.semwebtech.org/mondial/10/islands/Sicilia -qf alltriples.sparq
select ?X ?P ?Y
where {?X ?P ?Y}
```
[Filename: RDF/alltriples.sparql]

    (this way, alltriples.sparql can be applied to any LOD URL)

LOD: SPARQL Endpoints

LOD sources provide also a SPARQL protocol endpoint.

- optionally, many LOD endpoints provide an interactive SPARQL (HTML) Web interface:
  Mondial: via `http://www.semwebtech.org/mondial/10/`

Recall that SPARQL is also a communication *protocol* (for exchanging RDF data):

- W3C Recommendation at `http://www.w3.org/TR/sparql11-protocol/`
- usually at the URL `http://.../sparql`; GET, POST via HTTP
- GET via browser (query must be URL-encoded) shows formatted HTML table:
  `http://www.semwebtech.org/mondial/10/sparql?query=select%20%3fX%20where%20%7b%3fX%20a%20%3chttp%3a%2f%2fwww.semwebtech.org%2fmondial%2f10%2fmeta%23Country%3e%7d`
  (URL-encoding service at `https://www.branah.com/unicode-converter`)
- GET/POST from an RDF consumer:
  - result in SPARQL Query Results XML Format (set of tuples of bindings in XML)
    (described at `https://www.w3.org/TR/rdf-sparql-XMLres/`)
  - example (via browser): `http://rdf.insee.fr/sparql`, set response format to "RDF/XML" (actually, it is not RDF/XML, but SPARQL Query Results XML Format).

LOD Example: DBpedia

- data extracted from Wikipedia (since 2007)
- research project for Web data extraction and natural language processing
- data correctness: rather bad
- `http://dbpedia.org/page/France` Web page describing the RDF data about France;
- `http://dbpedia.org/data/France` delivers RDF/XML data about France (even to the browser); this can easily be parsed and queried by tools.
- SPARQL endpoint and Web interface at `http://dbpedia.org/sparql/`
- The URIs in dbpedia are actually of the form `http://dbpedia.org/resource/France` which redirects browsers to the HTML, and RDF tools to the RDF/XML data URL.

```
select ?X ?P ?Y
from <http://dbpedia.org/resource/France>
where {?X ?P ?Y}
```
[Filename: RDF/dbpedia-france.sparql]

```
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?X ?Y from <http://dbpedia.org/resource/France>
where {?X owl:sameAs ?Y}
```
[Filename: RDF/dbpedia-france-same-as.sparql]

- ... yields France owl:sameAs `http://sws.geonames.org/3017382/`.

LOD Example: Yago

"Yet Another Great Ontology"

- Project homepage: `https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`
- data extracted from Wikipedia (started 2007)
- research project for Web data extraction and natural language processing
- partially using and cleaning DBpedia base data
- quality/completeness: better than DBpedia
- resource URIs: `http://yago-knowledge.org/resource/Denmark` implemented (HTML) via redirection to SPARQL DESCRIBE queries of the form `http://lod.openlinksw.com/describe/?url=http%3A%2F%2Fdbpedia.org%2Fresource%2FDenmark`
- SPARQL endpoint and Web interface at `https://linkeddata1.calcul.u-psud.fr/sparql`
- Yago contains owl:sameAs-Links to Wikidata

LOD Example: geonames

- `http://sws.geonames.org/3017382/` in the browser: annotated google map with France;
- `http://sws.geonames.org/3017382/about.rdf` delivers RDF/XML data about France (For RDF, about.rdf is the equivalent to HTML's index.html)
- URL access to plain `http://sws.geonames.org/3017382/` with a *well-configured (see below)* RDF tool yields the triples:

```
select ?X ?P ?Y
from <http://sws.geonames.org/3017382/>
where {?X ?P ?Y}
```

[Filename: RDF/geonames-france.sparql]

- "rapper" tool: `rapper http://sws.geonames.org/3017382/`
- `http://sws.geonames.org/3017382/` is the resource describing France; while
- the triple <http://sws.geonames.org/3017382/about.rdf> <http://purl.org/dc/terms/modified> "2018-02-06"^^<http://www.w3.org/2001/XMLSchema#date> describes the *document* that describes France.

Many LOD sources are provided by e-Government Institutions

- insee.fr (L'Institut national de la statistique et des études économiques, France):
  - `http://rdf.insee.fr/`
  - Browser: HTML table that shows RDF triples, e.g.
    `http://id.insee.fr/geo/departement/42`
  - RDF access: request triples
    `rapper http://id.insee.fr/geo/departement/42`
  - SPARQL endpoint with Web interface at `http://rdf.insee.fr/sparql`
- United Kingdom Ordnance Survey: `https://data.ordnancesurvey.co.uk/`
  - City of Southampton: `https://data.ordnancesurvey.co.uk/doc/7000000000037256`
  - SPARQL:
    `https://data.ordnancesurvey.co.uk/datasets/boundary-line/explorer/sparql`
- Ordnance Survey Ireland: `https://osi.ie/blog/linked-data/`

Other LOD sources

- Wikidata: uses a specific modeling, see Slides 326 ff.
- overview of existing LOD data sources: `www.lod-cloud.net`

Aside: URL- “percent”-encoding

- whitespace $\rightarrow$ %20; some services allow “+” for whitespace.
- reserved symbols in URL structure: : / ? # [ ] @ ! $ & ( ) * + , ; =
  must be URL-encoded when occurring in a parameter string, if they could be misunderstood as URI structuring symbols
- currently for Mondial (and Insee.fr) , also `< > { }` must be encoded, “+” is not allowed for whitespace
- in SPARQL, there is the `ENCODE_FOR_URI`(*string*) function
  (I currently see no application for it *inside* SPARQL)

LOD: HTTP Accept Header

If an HTML interface is provided, LOD URLs are both “HTML URLs” and “LOD RDF URLs”:

- For HTML consumers (browsers), HTML should be delivered;
- For RDF consumers, RDF/XML, RDF/A (HTML with RDF annotations), Turtle unicode format etc. should be delivered.

$\Rightarrow$ HTTP Request contains appropriate information in the "accept" header:

- Browsers e.g.: text/html,application/xhtml+xml,
- XML consumers in general: application/xml,
- RDF consumers: application/rdf+xml, text/rdf, ... and some more.

- Technical observation
  - when running the jena-based tool from command line, it obtains triples;
  - when running it inside the Web Service, it obtained HTML (and threw an error)

  $\Rightarrow$ don’t access LOD data via simple GET *url*, but explicitly opening of HTTP connections with explicit
  ACCEPT: {text/turtle|application/rdf+xml} header.

Aside: Technical details of Querying a SPARQL Endpoint by HTTP+SPARQL Protocol

- See `https://www.w3.org/TR/sparql11-protocol/#query-operation`:
- HTTP GET,
- HTTP POST with URL-encoded parameters in message body,
- HTTP POST with unencoded SPARQL query in message body;
- query,
- optionally default graph and named graphs may be specified;
- accept media types for answer:
  SELECT query (variable bindings): application/sparql-results+xml
  CONSTRUCT query (graph/triples): application/rdf+xml or text/turtle
  DESCRIBE query (graph/triples): application/rdf+xml or text/turtle
- Not every SPARQL server supports each of these access methods;
- wget and curl as quick-and-dirty UNIX tools for sending HTTP requests;
- Java sample code.

Samples and tests: HTTP GET with parameters

- GET URLs contain the parameters in url-encoded form:

```
## wget -O -   outputs to stdout.
## optionally add    --header='Accept: application/sparql-results+xml'

wget -O - \
  http://dbpedia.org/sparql?query=select+distinct+?X+where+{?Y+a+?X}
wget -O - \
  http://id.insee.fr/sparql?query=select+distinct+?X+where+{?Y+a+?X}
     ## insee: wget: ok; GET URL via firefox must be URL-encoded (same effect as mond
  http://id.insee.fr/sparql?query=
    select%20distinct%20%3fX%20where%20%7b%3fY%20a%20%3fX%7d

## curl: {a,b,c} has a special semantics for curl, so encode it or turn {..} off:
curl http://dbpedia.org/sparql?query=select+distinct+?X+where+%7B?Y+a+?X%7D
curl -g http://dbpedia.org/sparql?query=ask{}
curl http://id.insee.fr/sparql?query=select+distinct+?X+where+%7B?Y+a+?X%7D
curl -g http://id.insee.fr/sparql?query=ask{}
```

- results are in SPARQL Query Results XML Format.

Result: SPARQL Query Results XML Format

```
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
 <head>
  <variable name="Concept"/>
 </head>
 <results distinct="false" ordered="true">
  <result>
   <binding name="Concept"><uri>http://www.w3.org/2002/07/owl#Thing</uri></binding>
  </result>
  <result>
   <binding name="Concept"><uri>http://www.w3.org/2002/07/owl#Class</uri></binding>
  </result>
  <result>
   <binding name="Concept"><uri>http://dbpedia.org/ontology/Person</uri></binding>
  </result>
  <result>
   <binding name="Concept"><uri>http://dbpedia.org/ontology/Country</uri></binding>
  </result>
   :
  and many more
   :
</sparql>
```

Samples: HTTP POST with URL-encoded parameters in message body

- POST: send parameters url-encoded in body

```
## wget: --method=POST must not be specified, if --post-data is used
## not necessary: --header='Content-Type: application/x-www-form-urlencoded' \

wget -O - \
     --post-data='query=select+distinct+?X+where+{?Y+a+?X}' \
     http://dbpedia.org/sparql

wget -O - \
     --post-data='query=select+distinct+?X+where+{?Y+a+?X}' \
     http://id.insee.fr/sparql

###curl -d: => POST;   -X POST not necessary
## curl optional: -H 'Content-Type: application/x-www-form-urlencoded'
curl -d 'query=select+distinct+?X+where+{?Y+a+?X}' \
     http://id.insee.fr/sparql
## equivalent:
curl --data-urlencode 'query=select distinct ?X where {?Y a ?X}' \
     http://id.insee.fr/sparql
```

Samples: HTTP POST with URL-encoded parameters in message body (cont'd)

```
### lotico.com delivers result as [application/sparql-results+json]:
### (lotico.com hosts the SPARQL protocol interface of http://www.geosparql.org/)

wget -O - \
     --post-data='query=select+distinct+?X+where+{?Y+a+?X}' \
     http://www.lotico.com:3030/lotico/sparql

### request results as xml:
wget -O - \
     --post-data='query=select+distinct+?X+where+{?Y+a+?X}' \
     --header='Accept: application/sparql-results+xml' \
     http://www.lotico.com:3030/lotico/sparql

curl --data-urlencode 'query=select distinct ?X where {?Y a ?X}' \
     http://www.lotico.com:3030/lotico/sparql

curl --data-urlencode 'query=select distinct ?X where {?Y a ?X}' \
     --header 'Accept: application/sparql-results+xml' \
     http://www.lotico.com:3030/lotico/sparql
```

Samples: HTTP POST with unencoded SPARQL query in message body

- HTTP POST with unencoded SPARQL query in message body;
  (some services do not support this)
- parameters (default graph etc.) still as parameters

```
### not supported by http://dbpedia.org/sparql and http://id.insee.fr/sparql

wget -O - --method=POST \
     --header='Content-Type: application/sparql-query' \
     --body-data='select distinct ?X where {?Y a ?X}' \
     http://www.lotico.com:3030/lotico/sparql

curl -d 'select distinct ?X where {?Y a ?X}' \
   -H 'Content-Type: application/sparql-query' \
   http://www.lotico.com:3030/lotico/sparql
```

Querying a SPARQL Endpoint by Java (SPARQL Protocol) – GET

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
public class SparqlGET {
  public static void main(String[] args) { try {
    BufferedReader br = null;
    URL inputURL = new URL("http://dbpedia.org/sparql?query=" +
          "select+distinct+?X+where+{?X+a+<http://dbpedia.org/ontology/Country>}");
    HttpURLConnection con = (HttpURLConnection) inputURL.openConnection();
    con.setRequestMethod("GET");
    con.setRequestProperty("Accept", "application/sparql-results+xml");
    con.connect();
    String s = "";   StringBuffer res= new StringBuffer();
    br = new BufferedReader(new InputStreamReader(con.getInputStream(), "UTF-8"));
    while ((s = br.readLine()) != null) { res.append(s+ "\n"); }
    br.close();
    System.out.println(res);  // or fill XML from Reader
  } catch (Exception e) { e.printStackTrace(); } }}
```

[Filename: RDF/SparqlGET.java]

Querying a SPARQL Endpoint by Java (SPARQL Protocol) – encoded POST

```
import java.io.BufferedReader; import java.net.HttpURLConnection; import java.net.URL;
import java.io.InputStreamReader;  import java.io.OutputStreamWriter;
public class SparqlEncPOST {
  public static void main(String[] args) { try {  BufferedReader br = null;
    URL inputURL = new URL("http://dbpedia.org/sparql");
    String q="query=construct+{+?X+a+<foo:country>}" +
             "+where+{?X+a+<http://dbpedia.org/ontology/Country>}";
    HttpURLConnection con = (HttpURLConnection) inputURL.openConnection();
    con.setRequestMethod("POST");
    con.setDoOutput(true);
    con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
    con.setRequestProperty("Accept", "text/turtle");
    con.connect();
      OutputStreamWriter wr = new OutputStreamWriter(con.getOutputStream());
      wr.write(q); wr.flush(); wr.close();
    String s = "";   StringBuffer res= new StringBuffer();
    br = new BufferedReader(new InputStreamReader(con.getInputStream(), "UTF-8"));
    while ((s = br.readLine()) != null) { res.append(s+ "\n"); }
    br.close();  System.out.println(res);  // or fill XML from Reader
  } catch (Exception e) { e.printStackTrace(); } }}
```

[Filename: RDF/SparqlEncPOST.java]

Querying a SPARQL Endpoint by Java (SPARQL Protocol) Direct POST

```
import java.io.BufferedReader; import java.net.HttpURLConnection; import java.net.URL;
import java.io.InputStreamReader;  import java.io.OutputStreamWriter;
public class SparqlDirectPOST {
  public static void main(String[] args) { try {
    BufferedReader br = null;
    URL inputURL = new URL("http://www.lotico.com:3030/lotico/sparql");
    String q="select distinct ?C where {?X a ?C}";
    HttpURLConnection con = (HttpURLConnection) inputURL.openConnection();
    con.setRequestMethod("POST");
    con.setDoOutput(true);
    con.setRequestProperty("Content-Type", "application/sparql-query");
    con.setRequestProperty("Accept", "application/sparql-results+xml");
      OutputStreamWriter wr = new OutputStreamWriter(con.getOutputStream());
      wr.write(q); wr.flush(); wr.close();
    con.connect();
    String s = "";   StringBuffer res= new StringBuffer();
    br = new BufferedReader(new InputStreamReader(con.getInputStream(), "UTF-8"));
    while ((s = br.readLine()) != null) { res.append(s+ "\n"); }
    br.close();   System.out.println(res);  // or fill XML from Reader
  } catch (Exception e) { e.printStackTrace(); } }}
```

[Filename: RDF/SparqlDirectPOST.java]

LOD: Classes and Properties as Resources

- The RDFS and OWL metadata about classes and properties is also accessible.
- it is left to the owner of a service what is delievered then.
- Access to `http://dbpedia.org/ontology/Country`
  (or rapper http://dbpedia.org/ontology/Country)
  - owl:equivalentClass, rdfs:subClassOf statements belong here.
- Access to `http://dbpedia.org/ontology/name` or
  `http://rdf.insee.fr/def/demo#population`
  - owl:equivalentProperty, rdfs:domain, rdfs:range statements belong here.
- Mondial: there are a lot of (OWL) descriptions that cannot be described in a single triple and that use blank nodes
  - HTML access to classes or properties: triples about the resource and all its instances:
    `http://www.semwebtech.org/mondial/10/meta%23locatedAt`
    `http://www.semwebtech.org/mondial/10/meta%23Water`
  - RDF: all triples of the `mondial-meta.n3` file, including the blank nodes.
  - RDF access to base url `http://www.semwebtech.org/mondial/10/`:
    all is-a-triples.

LOD: SPARQL Service Descriptions

- The SPARQL Recomendation also defines "Service Descriptions" (of the SPARQL endpoints) at
  `https://www.w3.org/TR/2013/REC-sparql11-service-description-20130321/`:

```
> rapper http://dbpedia.org/sparql

<http://dbpedia.org/sparql> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
     <http://www.w3.org/ns/sparql-service-description#Service>;
  <http://www.w3.org/ns/sparql-service-description#endpoint> <http://dbpedia.org/sparql>;
  <http://www.w3.org/ns/sparql-service-description#feature>
     <http://www.w3.org/ns/sparql-service-description#UnionDefaultGraph>;
  <http://www.w3.org/ns/sparql-service-description#feature>
     <http://www.w3.org/ns/sparql-service-description#DereferencesURIs>;
  <http://www.w3.org/ns/sparql-service-description#resultFormat>
     <http://www.w3.org/ns/formats/SPARQL_Results_JSON>;
  <http://www.w3.org/ns/sparql-service-description#resultFormat>
     <http://www.w3.org/ns/formats/SPARQL_Results_XML>;
  <http://www.w3.org/ns/sparql-service-description#resultFormat> <http://www.w3.org/ns/formats/Turtle>;
  <http://www.w3.org/ns/sparql-service-description#resultFormat> <http://www.w3.org/ns/formats/N-Triples>
  <http://www.w3.org/ns/sparql-service-description#resultFormat> <http://www.w3.org/ns/formats/N3>;
  <http://www.w3.org/ns/sparql-service-description#resultFormat> <http://www.w3.org/ns/formats/RDF_XML>;
  <http://www.w3.org/ns/sparql-service-description#supportedLanguage>
     <http://www.w3.org/ns/sparql-service-description#SPARQL10Query>;
  <http://www.w3.org/ns/sparql-service-description#url> <http://dbpedia.org/sparql> .
```

SPARQL 1.1: Queries against LOD Services

- SPARQL subqueries against SPARQL Web Services:
  `SERVICE` *url* `{` *pattern* `}`
  `SERVICE` *url* `{ SELECT DISTINCT` *variables* `WHERE {` *pattern* `} }`
  - In the first case, *all* tuples of bindings of all variables bound in *pattern* are sent back.
  - The second variant applies a projection on the remote service first.
- see `https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/`
- error if the remote service does not answer;
- SERVICE SILENT: empty result in case of an error.

```
select distinct ?X
where { service <http://dbpedia.org/sparql>
        {?X a <http://dbpedia.org/ontology/Country> . } }
```
[Filename: RDF/lodsparql.sparql]

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select distinct ?X ?L
where { service <http://www.semwebtech.org/mondial/10/sparql>
        {?X a :Country; :localname ?L. } }
```
[Filename: RDF/lodmondial.sparql]

VALUES clause (mainly used internally)

Consider

```
select ?X ?Y ?Z
where { pattern1(?X,?Y)
        service <url> { remote_pattern/query(?Y,?Z) }
        pattern2(?X,?Y,?Z) }
```

(Possible) internal evaluation:

- evaluate *pattern*$_1$(?X,?Y),
- call *url* to evaluate *remote_pattern*(?Y,?Z) only for the values for ?Y that have been collected by *pattern*$_1$(?X,?Y), binds additionally ?Z,
  (“sideways information passing”)
- evaluate *pattern*$_2$(?X,?Y,?Z)

The remote service will receive a query

```
select *
where { remote_pattern(?Y,?Z) .
        values ?Y { y1 y2 ... yn } }
```

VALUES clause constrains bindings

- although intended for peer-to-peer communication, the VALUES clause is also allowed for user queries:

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select distinct ?C ?CN
from <file:mondial.n3>
where { ?C a :Country; :carCode ?CC; :name ?CN .
        values ?CC { 'D' 'F' 'B' } }
```

[Filename: RDF/values-in.sparql]

- note the similarity with SQL's "`WHERE ... IN (...)`":

  SELECT name FROM country WHERE code IN ('D', 'F', 'B')

Distributed Queries - Mondial and insee.fr

Note: Distributed queries work only with the LOD SPARQL service at `http://www.semwebtech.org/mondial/10`; in the old /semweb one, ARQ destroys the syntax of the embedded subquery!

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
prefix insee: <http://rdf.insee.fr/def/geo#>
select distinct ?XN ?I ?DN ?PNL
from <file:mondial.n3>
where { ?C a :Country; :carCode 'F'; :name ?CN ; :hasCity ?X.
        ?P a :Province; :hasCity ?X .
        ?X :name ?XN.
        ?P :name ?PN .
        bind (strlang(?XN,"fr") as ?XNL) .
        bind (strlang(?PN,"fr") as ?PNL) .
        ### optional
        { service <http://rdf.insee.fr/sparql>
          { ?I a insee:Commune; insee:nom ?XNL; insee:subdivisionDe ?DEPT .
            ?DEPT a insee:Departement;  insee:nom ?DN; insee:subdivisionDe ?REG .
            ?REG insee:nom ?PNL
      } } }
```

[Filename: RDF/distributed-insee.sparql]

Distributed Queries: Allround Testcase

- SERVICE pattern can occur in classical inner join, optional/outer join and in FILTER NOT EXISTS/antijoin
- common variable is ?CN
- note: ?X produces ?CN-duplicates (that must be eliminated before submitting to remote service)

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select distinct ?C ?CN ?CC # ?X
where { ?C a :Country; :name ?CN ; :hasCity ?X .
        ## optional
        ## filter not exists
        { # service <http://localhost:8080/mondial/10/sparql>
          service <http://www.semwebtech.org/mondial/10/sparql>
          { ?CC a :City; :name ?CN }
      }
}
```

[Filename: RDF/distributed-testcase.sparql]

Distributed Queries - Mondial and insee.fr

- Variable in filter with equals
- note: (original) Jena does not communicate variables in a filter [3.2019]

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
prefix insee: <http://rdf.insee.fr/def/geo#>
select distinct ?XN ?I ?DN ?PNL
from <file:mondial.n3>
where { ?C a :Country; :carCode 'F'; :name ?CN ; :hasCity ?X.
        ?P a :Province; :hasCity ?X .
        ?X :name ?XN.
        ?P :name ?PN .
        ### optional
        { service <http://rdf.insee.fr/sparql>
          { ?I a insee:Commune; insee:nom ?XNL; insee:subdivisionDe ?DEPT .
            ?DEPT a insee:Departement;  insee:nom ?DN; insee:subdivisionDe ?REG .
            ?REG insee:nom ?PNL .
            FILTER (str(?XNL) = ?XN && str(?PNL) = ?PN)
          }
      }
}
```

[Filename: RDF/distributed-filter-equals.sparql]

Distributed Queries Testcase: Variable in Filter

- Variable in filter with numeric comparison
- note: Jena does not communicate variables in a filter

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
prefix insee: <http://rdf.insee.fr/def/geo#>
prefix inseeD: <http://rdf.insee.fr/def/demo#>
select distinct ?PN ?PN ?Pop ?XNL ?Pop2
from <file:mondial.n3>
where { ?C a :Country; :carCode 'F'; :name ?CN ; :hasProvince ?P .
        ?P :name ?PN; :population ?Pop . bind (strlang(?PN,"fr") as ?PNL) .
        { service <http://rdf.insee.fr/sparql>
          { ?I a insee:Commune; insee:nom ?XNL; insee:subdivisionDe ?DEPT .
            ?DEPT a insee:Departement;  insee:nom ?DN; insee:subdivisionDe ?REG .
            ?REG insee:nom ?PNL .
            ?I inseeD:population ?PopItem .
            ?PopItem inseeD:date ?D; inseeD:populationTotale ?Pop2 .
            FILTER (year(?D)=2015 && ?Pop2 > 200000 && ?Pop2 > 0.05 * ?Pop)
      } } }
```

[Filename: RDF/distributed-filter-comp.sparql]

- note: the filter is not “filter-safe” according to Slide 164
- “Push-into” to make a subquery self-contained is not applicable in the distributed case!

Distributed Queries Testcase: Service in NOT EXISTS

- Variable in filter with numeric comparison
- note: Jena 3.10 does not communicate variables in a filter

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
prefix insee: <http://rdf.insee.fr/def/geo#>
prefix inseeD: <http://rdf.insee.fr/def/demo#>
select distinct ?PN ?Pop
from <file:mondial.n3>
where { ?C a :Country; :carCode 'F'; :name ?CN ; :hasProvince ?P .
        ?P :name ?PN; :population ?Pop . bind (strlang(?PN,"fr") as ?PNL) .
        FILTER NOT EXISTS
        { service <http://rdf.insee.fr/sparql>
          { ?I a insee:Commune; insee:nom ?XNL; insee:subdivisionDe ?DEPT .
            ?DEPT a insee:Departement;  insee:nom ?DN; insee:subdivisionDe ?REG .
            ?REG insee:nom ?PNL .
            ?I inseeD:population ?PopItem .
            ?PopItem inseeD:date ?D; inseeD:populationTotale ?Pop2 .
            FILTER (year(?D)=2015 && ?Pop2 > 200000 && ?Pop2 > 0.05 * ?Pop)
      }}  }
```

[Filename: RDF/distributed-minus-service.sparql]

Distributed Queries: Trivial Testcase

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
SELECT ?c ?name
WHERE {
   ?c a :Continent.
   ## SERVICE <http://localhost:8080/mondial/10/sparql>{
   SERVICE <http://www.semwebtech.org/mondial/10/sparql>{
   ?c :name ?name.
   }
}
```

[Filename: RDF/distributed-trivial.sparql]

Distributed Queries: Variable Visibility Testcase

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?C ?N
where { ?C a :Country ; :name ?N .
        service <http://localhost:8080/mondial/10/sparql>
          { select distinct ?C   ## ?N
            where
            { ?C :hasCity ?X . ?X :name ?N
          } }
}
```

[Filename: RDF/service-projection-test-mondial.sparql

- the inner ?N is logically local, *not* to bei a join condition with the outer one.
- correct eval: 244 countries.
- wrong eval: 10 countries (having a city name = country name)
- original Jena 3.10: correct eval.

Distributed Queries - Mondial and wikidata

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix : <http://www.semwebtech.org/mondial/10/meta#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
select distinct ?C ?CN ?label
from <file:mondial.n3>
from <file:mondial-sameas.n3>
where { ?C a :Country;  :name ?CN ; :hasCity ?X.
        ?X :name ?XN; owl:sameAs ?URL
        SERVICE <https://query.wikidata.org/sparql>
          {
            ?URL wdt:P31/wdt:P279* wd:Q7930989 .
            ?URL wdt:P17 ?wc .    ### (in) country
            ?wc rdfs:label "France"@en .
            ?URL rdfs:label ?label.
            FILTER (langMatches( lang(?label), "fr")) .
          }
       ### values ?url ( ...)
}
```

[Filename: RDF/distributed-wikidata.sparql]

- inner query (wikidata) 91 results (in France; 49000 city/towns worldwide)
- outer query (mondial cities with sameAs): 1300/1800
- intersection: 16 cities
  - eval outer (1ms), get 1300 results, state 1300 individual queries with ?URL replaced (each 0.3s) $\rightarrow$ 400sec.
  - eval outer, get 1300 results, state inner query with 1300 values (??sec)
  - eval outer (1ms), eval inner (1.4 sec), join (1ms)

Chapter 8
Knowledge Graph: Statements about Statements etc

Sometimes, statements/triples need annotations, (temporal) qualifications, references etc.

- Reification as a concept in RDF: Statements
- Complex representation of knowledge based on plain RDF

8.1 Further RDF Vocabulary: Reification

Take statements (=triples) as resources and make statements about them:

- `rdf:Statement` which has properties `rdf:subject`, `rdf:predicate`, `rdf:object`, that yield a resource.
- RDF/XML: give an ID to the statement.
  "The statement "Germany had 83536115 inhabitants" was valid in year 1997":

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:mon="http://www.semwebtech.org/mondial/10/meta#"
     xml:base="http://www.semwebtech.org/mondial/10/">
  <mon:Country rdf:about="countries/D">
    <mon:name>Germany</mon:name>
    <mon:population rdf:ID="de-pop">83536115</mon:population>
  </mon:Country>
  <rdf:Description rdf:about="#de-pop">
    <mon:year>1997</mon:year> </rdf:Description>
</rdf:RDF>
```

[Filename: RDF/reification.rdf]

(note the usage of rdf:ID for individuals (Slide 260), for classes and properties (Slide 275), and here for *Statements*)

REIFICATION: EXAMPLE

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:mon="http://www.semwebtech.org/mondial/10/meta#"
     xml:base="http://www.semwebtech.org/mondial/10/">
  <mon:Country rdf:about="countries/D">
    <mon:name>Germany</mon:name>
    <mon:population rdf:ID="de-pop">83536115</mon:population>
  </mon:Country>
  <rdf:Description rdf:about="#de-pop">
    <mon:year>1997</mon:year> </rdf:Description>
</rdf:RDF>
```

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
select ?X ?P ?V
from <file:reification.rdf>
where {?X a rdf:Statement; ?P ?V}
```

[Filename: RDF/reification.sparql]

Triples (added automatically by the RDF semantics)

(use `jena -t -if reification.rdf` or see RDF validator):

(<http://.../mondial/10/#de-pop> rdf:type rdf:Statement)

(<http://.../mondial/10/#de-pop> rdf:subject <http://.../mondial/10/countries/D>)

(<http://.../mondial/10/#de-pop> rdf:predicate <http://.../mondial/10/meta#population>)

(<http://.../mondial/10/#de-pop> rdf:object “83536115”)

(<http://.../mondial/10/#de-pop> <http://.../mondial/10/meta#year> “1997”)

... the above annotates a statement that is assumed to hold.

Reification in the Semantic Web

Annotating Statements:

- with probabilities, trust, “who says ...”, even negation!
- annotations can be in different files than the annotated statements (cf. out-of-line XLink arcs),
- can be used for reasoning.

Note:

- information about a *predicate* (which describes the predicate “name” (e.g., the source where all this data is taken from, transitivity or symmetry) or the set of all instances (cardinalities), or each its (range, domain))

is different

- from describing/annotating a *statement* (i.e. one instance of a predicate).

Reification and Annotation

- Statements that do *not* hold can also by annotated:

```
<!DOCTYPE rdf:RDF [ <!ENTITY mon "http://www.semwebtech.org/mondial/10/meta#">
                    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#"> ]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:mon="http://www.semwebtech.org/mondial/10/meta#"
     xml:base="http://www.semwebtech.org/mondial/10/">
  <rdf:Statement rdf:ID="cap-d-bonn">  <!-- ********************* BONN ***************** -->
    <rdf:subject rdf:resource="countries/D/"/>
    <rdf:predicate rdf:resource="&mon;capital"/>
    <rdf:object rdf:resource="countries/D/provinces/NordrheinWestfalen/cities/Bonn"/>
    <mon:from rdf:datatype="&xsd;date">1949-11-29</mon:from>
    <mon:until rdf:datatype="&xsd;date">1990-10-02</mon:until>
  </rdf:Statement>
  <rdf:Description rdf:about="countries/D/"> <!-- *************** BERLIN *************** -->
   <mon:capital rdf:ID="cap-d-berlin"
     rdf:resource="countries/D/provinces/Berlin/cities/Berlin"/>
  </rdf:Description>
  <rdf:Description rdf:about="#cap-d-berlin">
    <mon:from rdf:datatype="&xsd;date">1990-10-03</mon:from>
  </rdf:Description>
</rdf:RDF>
```

[Filename: RDF/reification-2.rdf]

- note (RDF/XML): attribute URL values in rdf:resource and rdf:datatype are subject to expansion wrt. xml:base, but not wrt. namespaces, so monmeta:capital and xsd:date must be either typed completely or constructed with XML-entities.

Reification and Annotation (Cont'd)

- queries against the above information

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
select ?S ?X ?Y ?F ?U
from <file:reification-2.rdf>
where {{?X mon:capital ?Y} UNION
       {?S rdf:subject ?X ; rdf:predicate mon:capital; rdf:object ?Y
        OPTIONAL {?S mon:from ?F} . OPTIONAL {?S mon:until ?U}}}
```

[Filename: RDF/reification-2.sparql]

- (?X capital ?Y) contains only (Germany, Berlin), the *triple* that actually holds.
- The statement (Germany, capital, Bonn) is not a triple, but there is only annotation about it.

Annotated Knowledge Representation

- A knowledge base in a setting where annotations are important, thus would not contain any explicit triples but only described statements
- Applications must then interpret the semantics of the annotations:
  - heuristics to generate those triples that are assume to hold actually:
  - annotation of probabilities, opinion, provenance, trust, ...
  - if an RDF source contains a statement that is somewhere else annotated that it held only in earlier times, discard it,
  - if some statement does not hold, but is e.g. annotated as believed by a trusted person, consider it to be true.
- example: Wikidata, see next section.

8.2 Wikidata: a Linked Open Data Source

Wikidata uses an own modeling:

- query interface at `https://query.wikidata.org/`
- fun factor: low – wikidata poses a time limit on the evaluation of queries that can hardly be met even for rather simple examples
- not a “simple database” but a comprehensive, multilingual “knowledge graph”.
- uses just plain RDF data + rdfs:label, no rdfs:subClassOf, no OWL, no reasoning at all.
- cf. paradoxes (Slide 228) whenever something is both an instance and a class (e.g. “penguin” as a the class of all penguins, and an instance “penguin is a species”)
- everything (things/classes and also properties) has an rdfs:label in most of the wikipedia/wikidata languages
  $\Rightarrow$ properties/classes/things are not identified by their name (in any language), but just by a number
  (e.g. P36 is “has capital”, P31 means “instance of”, and P279 means “subclass of”)
- statements are intensively annotated
- class hierarchy *without inherent transitivity* (no reasoning, partially redundant storage)

Wikidata Things and Properties

The query interface `https://query.wikidata.org/` has predefined prefixes for a set of wikidata namespaces:

- wd: is the namespace for all wikidata things (entities and/or classes)

  ```
  PREFIX wd: <http://www.wikidata.org/entity/>
  ```

  URI of Germany: `https://www.wikidata.org/entity/Q183`
  When browsing, this is automatically redirected to
  `https://www.wikidata.org/wiki/Q183`

- wdt: is the wikidata namespace for "direct" properties (i.e., the application-level properties in real-world triples):

  ```
  PREFIX wdt: <http://www.wikidata.org/prop/direct/>
  ```

  URI of the "capital" property: `http://www.wikidata.org/prop/direct/P36`
  When browsing, this is automatically redirected to
  `https://www.wikidata.org/wiki/Property:P36`

- everything has rdfs:label-s in many languages.

Wikidata Classes and Class Hierarchy

```
SELECT DISTINCT ?c ?cn ?capn
WHERE {
   ?c wdt:P31 wd:Q6256.          ## instanceOf country
   filter not exists {?c wdt:P576 ?X}  ## dissolved date
   ?c rdfs:label ?cn.
   ?c wdt:P36  ?cap . ?cap rdfs:label ?capn  ## capital
   FILTER ((langMatches(lang(?cn),"en") && langMatches(lang(?capn),"en")) ||
           (langMatches(lang(?cn),"ru") && langMatches(lang(?capn),"ru"))    )
   ## FILTER (contains(str(?cn),'Germany'))
}
```

[Filename: RDF/wikidatacountrycaps1.sparql]

- wd:Q6256 (see `https://www.wikidata.org/wiki/Q6256`) has broad meaning, Q3624078 means “souvereign state” (still including ancient ones). (but with this, it runs into a server error; 14.1.2019)
- Germany does not belong (directly) to Q3624078.

Wikidata Classes and Class Hierarchy

- use P31 (instance of) and P279 (subclass of) in a path expression

```
SELECT DISTINCT ?c ?cn ?capn
WHERE {
   ?c wdt:P31/wdt:P279* wd:Q6256.   ## path: isa/subclass*
   filter not exists {?c wdt:P576 ?X}  ## dissolved date
   ?c rdfs:label ?cn.
   ?c wdt:P36  ?cap . ?cap rdfs:label ?capn  ## capital
   FILTER ((langMatches(lang(?cn),"en") && langMatches(lang(?capn),"en")) ||
           (langMatches(lang(?cn),"ru") && langMatches(lang(?capn),"ru"))    )
   ## FILTER (contains(str(?cn),'Germany'))
}
```

[Filename: RDF/wikidatacountrycaps2.sparql]

Basic Knowledge Graph + Multilingual labels

- the basic graph of the resources and statements exists only once, based on the wd:Qxxx (things), wdt:Pyyy (properties) and wds:zzz (statements),
- Qxxx and Pyyy have labels in each/most of the languages.

Wikidata: Built-In Handling for Labels and Languages

The service supports several extensions to standard SPARQL capabilities:
(see `https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual`

- for objects bound to a variable ?X, automatically bind ?XLabel (if SELECTed) to the label(s) in selected languages:

```
PREFIX wikibase: <http://wikiba.se/ontology#>
 SERVICE wikibase:label {
     bd:serviceParam wikibase:language "de,en" .
 }
```

```
SELECT DISTINCT ?c ?cLabel ?capLabel
WHERE {
   ?c wdt:P31/wdt:P279* wd:Q6256.   ## path: isa/subclass*; Q6256=country
   filter not exists {?c wdt:P31/wdt:P279* wd:Q3024240}  ## historical country
   filter not exists {?c wdt:P576 ?x}  ## dissolved date
   ?c wdt:P36  ?cap .  ## capital
SERVICE wikibase:label { bd:serviceParam wikibase:language "en,fr" }
}
```

[Filename: RDF/wikidatacountrycaps3.sparql]

Combination of Wikidata and Mondial/ SPARQL SERVICE clause

```
PREFIX : <http://www.semwebtech.org/mondial/10/meta#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?mcont ?wdcont ?labelWithoutLanguageTag
FROM <http://www.dbis.informatik.uni-goettingen.de/Mondial/Mondial-RDF/mondial.n3>
WHERE {
    SERVICE <https://query.wikidata.org/sparql>{
        ?wdcont wdt:P31 wd:Q5107.     ### instanceOf continent
        ?wdcont rdfs:label ?label.
        FILTER (langMatches( lang(?label), "*" ) )
        BIND(STR(?label) AS ?labelWithoutLanguageTag)
    }
    ?mcont a :Continent.
    ?mcont :name ?labelWithoutLanguageTag.
}
```

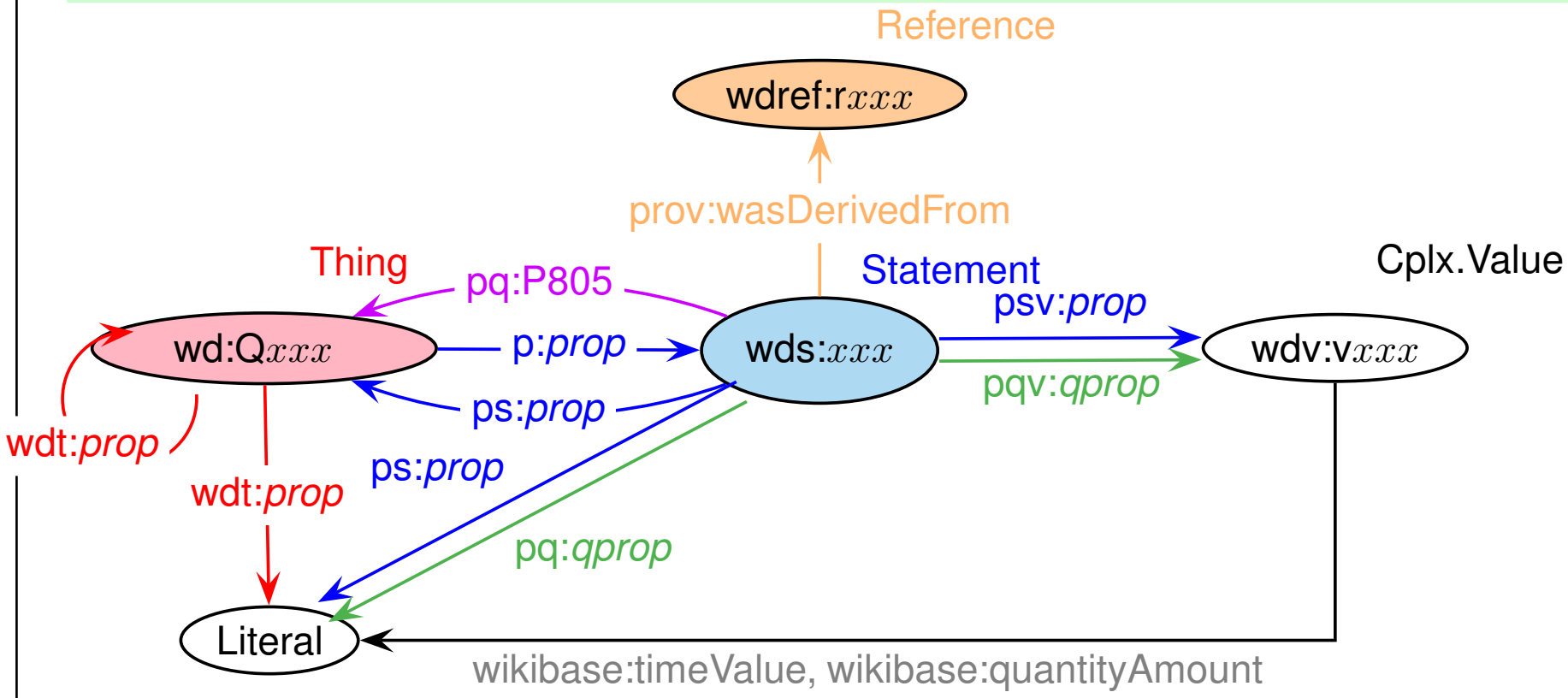
[Filename: RDF/lodwikidata.sparql]

Wikidata: Prefixes and types of edges and nodes (Graphics see next slide)

(see `https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format`)

- wdt:*prop*: "real-world" edges: object $\overset{prop}{\rightarrow}$ things | literals
- p:*prop*: thing $\rightarrow$ *prop*-statements (one statement for each filler of property *prop* of a thing)
- ps:*prop*: *prop*-statement $\rightarrow$ thing | literal
- psv:*prop*: *prop*-statement $\rightarrow$ cplxValue if the value is a complex one
- wikibase:timeValue ($\rightarrow$ xsd:date) and wikibase:quantityAmount ($\rightarrow$ numeric): cplxValue $\rightarrow$ literal
  wikibase:timeCalendarModel, wikibase:quantityUnit (units),
  wikibase:geoLatitude/geoLongitude (for type wikibase:GlobecoordinateValue)
- pq:*qprop*: qualifier properties: statement $\rightarrow$ literal
  (start, end, timepoint ... as xsd:date)
  pqv:*qprop*: qualifier properties: statement $\rightarrow$ cplxValue (time values)
- pq:P805: statement $\rightarrow$ Thing – "isSubjectOf", better name would be "is reified as"
  $\Rightarrow$ Thing p:*prop*/pq:P805 $\rightarrow$ Thing leads to reified real-world things.
- prov:wasDerivedFrom: statement $\rightarrow$ Reference – *Provenance*: references where the statement can be derived from.

Wikidata: Prefixes and types of edges and nodes



- wdt.*prop* = p:*prop* ∘ ps:*prop*
- wdt.*prop* = p:*prop* ∘ psv:*prop* ∘ (wikibase:timeValue | wikibase:quantityAmount) (complex values)
- properties pointing to objects that represent reified properties:
  wdt.*prop'* = p:*prop* ∘ pq:P805 ?? (Country: hasMembership = p:isMember ∘ pq:P805)

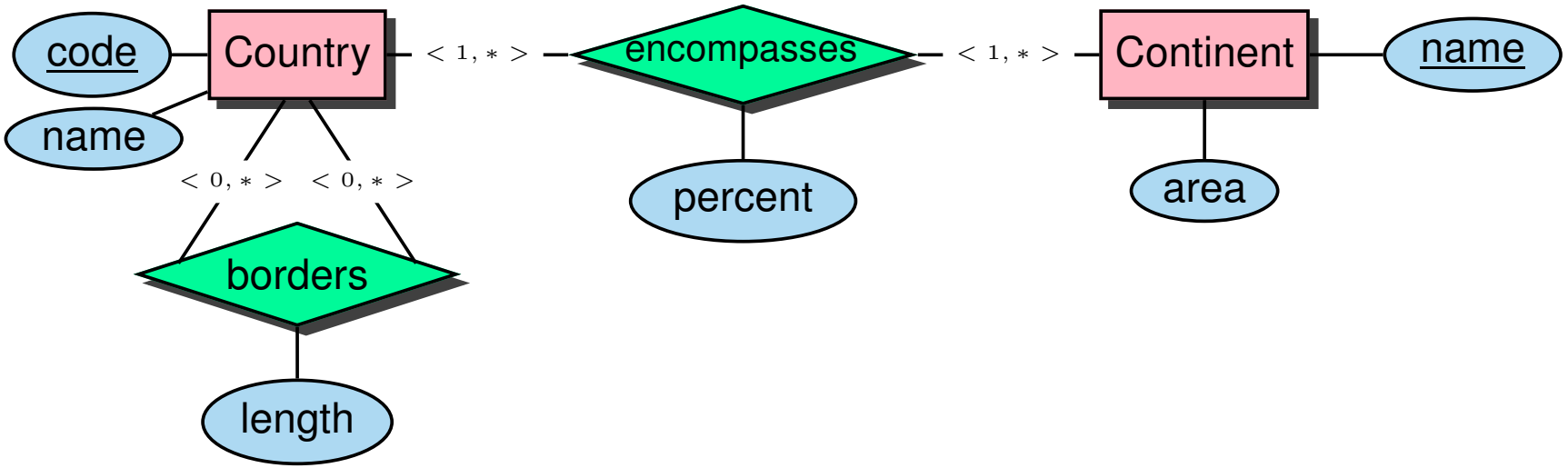
Wikidata: Temporal qualifiers: capitals

```
SELECT ?cl ?capl ?from ?fr ?until ?un
WHERE
{ ?c wdt:P31/wdt:P279* wd:Q3624078 . ## sovereign state
  FILTER NOT EXISTS { ?c wdt:P576 ?X }  ### not dissolved date
  ?c p:P36 ?stmt .       ## P36: capital
    ?stmt ps:P36 ?cap ;
          optional { ?stmt pq:P580 ?from }
          optional { ?stmt pq:P582 ?until } .
  ?c rdfs:label ?cl .
  ?cap rdfs:label ?capl .
  filter (lang(?cl)="en")
  filter (lang(?capl)="en")
  bind (str(?from) as ?fr) . bind (str(?until) as ?un)  ### as xsd:dates
  # filter (str(?cl) = "Germany")
}
order by ?cl ?fr
```

[Filename: RDF/wikidata-capital.sparql]

Wikidata: Properties with Properties – Reification

- Recall from DB lectures: the issue of *reification* of attributes occurs in many data models (UML, XML, ... and also in RDF)



- Consider that Germany borders Austria with a length of 815km.
- Wikidata: the Wikidata-Statement-Object has a “isSubjectOf” (better name would be “is reified as”) (pq:P805) property that points to a Wikidata resource (here, Q1991986) which is the “Austria-Germany border” and has P2043 (length) with a value of 815.

Wikidata: Reification qualifiers: border length

```
SELECT ?cl ?neighl ?border ?bl ?len ?len2
WHERE
{ ?c wdt:P31/wdt:P279* wd:Q3624078 . ## sovereign state
  FILTER NOT EXISTS { ?c wdt:P576 ?X }  ### not dissolved date
  ?c p:P47 ?stmt .        ## P47: shares border with
    ?stmt ps:P47 ?neighb .
  FILTER NOT EXISTS { ?neighb wdt:P576 ?Y }  ### not dissolved date
  ?c rdfs:label ?cl .
  ?neighb rdfs:label ?neighl .
  ?stmt pq:P805 ?border .   ### P805: isSubjectOf - "Reification"
  ?border rdfs:label ?bl ;
          wdt:P2043 ?len; p:P2043/ps:P2043 ?len2.  ### P2043: length
  filter (lang(?cl)="en")
  filter (lang(?neighl)="en")
  filter (lang(?bl)="en")
  ## filter (str(?cl) = "Germany")
}
order by ?cl ?neighl
```

[Filename: RDF/wikidata-borderlength.sparql]

Wikidata: countries encompassed by continents

```
SELECT DISTINCT ?cl ?contl ?PL ?VL ?reif
WHERE
{ ?c wdt:P31/wdt:P279* wd:Q3624078 . ## sovereign state
  FILTER NOT EXISTS { ?c wdt:P576 ?X }  ### not dissolved date
  ?c p:P30 ?stmt .        ## P30: continent
    ?stmt ps:P30 ?cont .
  ?c rdfs:label ?cl .
  ?cont rdfs:label ?contl .
  optional { ?stmt ?P ?V . ?P rdfs:label ?PL . ?V rdfs:label ?VL .
             filter (lang(?PL)="en")
             filter (lang(?VL)="en") }
  optional {?stmt pq:P805 ?reif} .   ### P805: isSubjectOf - "Reification"
  filter (lang(?cl)="en")
  filter (lang(?contl)="en")
}
order by ?cl ?contl
```

[Filename: RDF/wikidata-encompassed.sparql]

- no bindings for ?P, ?V, or ?reif.

$\Rightarrow$ no information apart from the pure “X is (partly) located on continent Y”

Wikidata: ethnic groups in countries

```
SELECT DISTINCT ?cl ?ethl ?PL ?VL ?reif
WHERE
{ ?c wdt:P31/wdt:P279* wd:Q3624078 . ## sovereign state
  FILTER NOT EXISTS { ?c wdt:P576 ?X }  ### not dissolved date
  ?c p:P172 ?stmt .       ## P172: ethnic group
      ?stmt ps:P172 ?eth .
  ?c rdfs:label ?cl .
  ?eth rdfs:label ?ethl .
  optional { ?stmt ?P ?V . ?P rdfs:label ?PL . ?V rdfs:label ?VL .
             filter (lang(?PL)="en")
             filter (lang(?VL)="en") }
  optional {?stmt pq:P805 ?reif} .   ### P805: isSubjectOf - "Reification"
  filter (lang(?cl)="en")
  filter (lang(?ethl)="en")
}
order by ?cl
```

[Filename: RDF/wikidata-ethnics.sparql]

- no bindings for ?P, ?V, or ?reif $\rightarrow$ no percentages.
- no information about languages and religions in a country

Wikidata: asking for property labels

Each property Pxxx "exists" in many different ways:

- wd:Pxxx is the property as an object – it has an rdfs:label
- all usages as wdt:Pxxx, p:Pxxx, ps:Pxxx etc do not have labels.
- for querying "return all properties (with names) of an object", the following connections between wd:Pxxx and the other usages must be used:

```
wd:Pxxx a wikibase:Property ;
    wikibase:directClaim wdt:Pxxx ;
    wikibase:claim p:Pxxx ;
    wikibase:statementProperty ps:Pxxx ;
    wikibase:statementValue psv:Pxxx ;
    wikibase:qualifier pq:Pxxx ;
    wikibase:qualifierValue pqv:Pxxx ;
    wikibase:reference pr:Pxxx ;
    wikibase:referenceValue prv:Pxxx ;
    wikibase:novalue wdno:Pxxx .
```

Wikidata: reified things

- this query lists all properties PL of countries whose statements are reified as objects (YL), and optionally gives the propertyname (x RPL y)

```
SELECT DISTINCT ?cl ?PL ?YL ?RPL1 ?RPL2  WHERE
{ ?c wdt:P31/wdt:P279* wd:Q3624078 . ## sovereign state
  FILTER NOT EXISTS { ?c wdt:P576 ?X }  ### not dissolved date
  ?c rdfs:label ?cl .  filter (lang(?cl)="en")
  filter (str(?cl) = "Germany")
  ?c ?P ?stmt . ?stmt pq:P805 ?Y .  ## p:P-stmt reified as Y
  ?Prop wikibase:claim ?P; rdfs:label ?PL . filter (lang(?PL)="en")
  ?Y rdfs:label ?YL .   filter (lang(?YL)="en")
  ### link between ?c and ?Y or back: must be in wdt:rprop
  optional { ?c ?rp1 ?Y .
             ?Rprop wikibase:directClaim ?rp1 .
             ?Rprop rdfs:label ?RPL1 . filter (lang(?RPL1)="en") }
  optional { ?Y ?rp2 ?c .
             ?Rprop wikibase:directClaim ?rp2 .
             ?Rprop rdfs:label ?RPL2 . filter (lang(?RPL2)="en") }
} order by ?cl
```

[Filename: RDF/wikidata-reified.sparql]

Answer e.g. : cl/Germany, PL/shares border with, YL/Austria-Germany border, RPL2/country

Wikidata: Temporal qualifiers/data quality

- quality is better than DBpedia, but not perfect:

```
SELECT *
WHERE
{ ?uni rdfs:label ?val .
  ?uni wdt:P31/wdt:P279* wd:Q3918 .     ## Q3918: University
  ?uni p:P17 ?stmt .       ## P17: located in country
    ?stmt ps:P17 ?C ;
          optional { ?stmt pq:P580 ?start }
          optional { ?stmt pq:P582 ?end } .
  ?C rdfs:label ?cl
  filter (lang(?cl)="en")
  filter (lang(?val)="en")
  filter (str(?val) = "Moscow State University")
}
```

[Filename: RDF/lomonossow.sparql]

- Countries: Russia, Soviet Union, Russian Empire, (Tajikistan - 10.1.2021)
  the “start” and “end” entries are often missing.

Chapter 9
Ontologies and the Web Ontology Language – OWL

- *vocabularies* can be defined by RDFS
  - not so much stronger than the ER Model or UML
    (even weaker: no cardinalities)
  - not only a conceptual model, but a “real language” with a close connection to the data level (RDF)
  - *incremental* world-wide approach
  - “global” vocabulary can be defined by autonomous partners
- but: still restricted when *describing* the vocabulary.

Ontologies/ontology languages further extend the expressiveness:

- Description Logics
- Topic Maps (in SGML) since early 90s, XTM (XML Topic Maps)
- Ontolingua – non-XML approach from the Knowledge Representation area
- OIL (Ontology Inference Layer): initiative funded by the EU programme for Information Society Technologies (project: On-To-Knowledge, 1.2000-10.2002); based on RDF/RDFS
- DAML (Darpa Agent Markup Language; 2000) ... first ideas for a Semantic Web language
- DAML+OIL (Jan. 2001)
- developed into OWL (1st version March 02, finalized Feb. 04)

THREE VARIANTS OF OWL

Several expressiveness/complexity/decidability levels:

- OWL Full: extension of RDF/RDFS
  - classes can also be regarded as individuals (have properties, classes of classes etc.)
- OWL DL
  - fragment of OWL that fits into the Description Logics Framework:
    - the sets of classes, properties, individuals and literals are disjoint

    $\Rightarrow$ only individuals can have arbitrary user-specified properties;
    classes and properties have only properties from the predefined RDFS and OWL vocabularies.
  - decidable reasoning
  - OWL 1.0 (2004), OWL 2.0 (2009)
- OWL Lite
  - subset of OWL DL
  - easier migration from frame-based tools (note: F-Logic is a frame-based framework)
  - easier reasoning (translation to Datalog)

9.1 Description Logics

- Focus on the description of *concepts*, not of instances
- Terminological Reasoning
- Origin of DLs: Semantic Networks (graphical formalism)

Notions

- Concepts (= classes),
  note: literal datatypes (string, integer etc.) are not classes in DL and OWL, but *data ranges*
  (cf. XML Schema: distinction between simpleTypes and complexTypes)
- Roles (= relationships),
- A Description Logic alphabet consists of a finite set of concept names (e.g. Person, Cat, LivingBeing, Male, Female, . . .) and a finite set of role names (e.g., hasChild, marriedTo, . . .),
- constructors for derived concepts and roles,
- axioms for asserting facts about concepts and roles.

Comparison with other Logics

Syntax and semantics defined different but similar from first-order logic

- formulas over an alphabet and a small set of additional symbols and combinators
- semantics defined via *interpretations* of the combinators
- set-oriented, no instance variables
  (FOL: instance-oriented with domain quantifiers)
- family of languages depending on what combinators are allowed.

The base: $\mathcal{AL}$

The usual starting point is $\mathcal{AL}$:

- "attributive language"
- Manfred Schmidt-Schauss and Gert Smolka: *Attributive Concept Descriptions with Complements*. In *Artificial Intelligence* 48(1), 1991, pp. 1–26.
- extensions (see later: $\mathcal{ALC}$, $\mathcal{ALCQ}$, $\mathcal{ALCQ}(D)$, $\mathcal{ALCQI}$, $\mathcal{ALCN}$ etc.)

Atomic, Named Concepts

- atomic concepts, e.g., Person, Male, Female
- the "universal concept" $\top$ (often called "Thing" – everything is an instance of Thing)
- the empty concept $\bot$ ("Nothing"). There is no thing that is an instance of $\bot$.

Concept Expressions using Set Operators

- intersection of concepts: $A \sqcap B$
  Adult $\sqcap$ Male
- negation: $\neg A$
  $\neg$Italian , Person $\sqcap \neg$Italian
- union (disjunctive concept): $A \sqcup B$
  Cat $\sqcup$ Dog – things where it is known that they are cats or dogs, but not necessarily which one.

Concept Expressions using Roles

Concepts (as an intensional characterization of sets of instances) can be described implicitly by their properties (wrt. *roles*).

Let R be a role, C a concept. Then, the expressions $\exists R.C$ and $\forall R.C$ also describe concepts (intensionally defined concepts) by constraining the roles:

- Existential quantification: $\exists R.C$ – all things that have a *filler* for the role R that is in C.
  $\exists$hasChild.Male means "all things that have a male child".
  Syntax: the whole expression is the "concept expression", i.e., $\exists$hasChild.Male(john) stands for ($\exists$hasChild.Male)(john).
- Range constraints: $\forall R.C$
  $\forall$hasChild.Male means "all things that have only male children (including those that have no children at all)".
- Note that $\bot$ can be used to express non-existence: $\forall R.\bot$: all things where all fillers of role R are of the concept $\bot$ (= Nothing) – i.e., all things that do not have a filler for the role R.
  $\forall$hasChild.$\bot$ means "all things that have no children".

Semantics of Concept Constructors

As usual: by interpretations.

An interpretation $\mathcal{I} = (I, \mathcal{D})$ consists of the following:

- a domain $\mathcal{D}$,
- for every concept name C: $I(C) \subseteq \mathcal{D}$ is a subset of the domain,
- for every role name R: $I(R) \subseteq \mathcal{D} \times \mathcal{D}$ is a binary relation over the domain.

Structural Induction

- $I(A \sqcup B) = I(A) \cup I(B)$
- $I(A \sqcap B) = I(A) \cap I(B)$
- $I(\neg A) = \mathcal{D} \setminus I(A)$
- $I(\exists R.C) = \{x \mid \text{there is an } y \text{ such that } (x, y) \in I(R) \text{ and } y \in I(C)\}$
- $I(\forall R.C) = I(\neg\exists R.(\neg C)) = \{x \mid \text{for all } y \text{ such that } (x, y) \in I(R),\ y \in I(C)\}$

Example

Male $\sqcap$ $\forall$hasChild.Male is the set of all men who have only sons.

STRUCTURE OF A DL KNOWLEDGE BASE

DL Knowledge Base

TBox (schema) – "Terminological Box"

Statements/Axioms about concepts

$\text{Man} \equiv \text{Person} \sqcap \text{Male}$

$\text{Parent} \equiv \text{Person} \sqcap (\exists \geq 1\ \text{hasChild}.\top)$

$\text{ParentOfSons} \equiv \text{Person} \sqcap (\exists \geq 1\ \text{hasChild.Male})$

$\text{ParentOfOnlySons} \equiv \text{Person} \sqcap (\forall\ \text{hasChild.Male})$

ABox (data) – "Assertional Box"

Statements/Facts about individuals

$\text{Person(john)},\ (\text{Adult} \sqcap \text{Male})(\text{john}),\ (\neg\text{Italian})(\text{john})$

hasChild(john,alice), age(alice,10), Female(alice)

hasChild(john,bob), age(bob,8), Male(bob)

$\forall\text{hasChild}.\bot\ (\text{alice}),\ \neg\exists\text{hasChild}.\top\ (\text{bob})$

THE TBOX: TERMINOLOGICAL AXIOMS

Definitions and assertions (to be understood as constraints, and as knowledge that can be used for deduction, e.g. of class membership) about concepts:

- concept subsumption: $C \sqsubseteq D$; defining a concept hierarchy.
  $\mathcal{I} \models C \sqsubseteq D \;\; :\Leftrightarrow \;\; I(C) \subseteq I(D)$.
- concept equivalence: $C \equiv D$; often used for defining the left-hand side concept.
  Semantics: $\mathcal{I} \models C \equiv D \;\; :\Leftrightarrow \;\; C \sqsubseteq D$ and $D \sqsubseteq C$.

TBox Reasoning

- is a concept C satisfiable?
- is $C \sqsubseteq D$ implied by a TBox?
- given the definition of a new concept D, classify it wrt. the given concept hierarchy.

THE ABOX: ASSERTIONAL AXIOMS

- contains the facts about instances (using names for the instances) in terms of the basic concepts and roles:
  Person(john), Male(john), hasChild(john,alice)
- contains also knowledge in terms of intensional concepts, e.g., ∃hasChild.Male(john)

TBox + ABox Reasoning

- check consistency between ABox and a given TBox
- ask whether a given instance satisfies a concept C
- ask for all instances that have a given property
- ask for the most specific concepts that an instance satisfies

Note: instances are allowed only in the ABox, not in the TBox.

If instances should be used in the definition of concepts (e.g., “European Country” or “Italian City”), *Nominals* must be used (see later).

THE BASIS: $\mathcal{AL}$

Concept expressions can be composed as follows:

- intersection of concepts, negation of *atomic* concepts: $C \sqcap D$, $\neg A$
- restricted existential quantification: $\exists R.\top$
  $\exists$hasChild.$\top$ means "all things that have a child (... that belongs to the concept "Thing")".
- universal restriction: $\forall R.C$
  $\forall$hasChild.Person means "if some thing is a "filler" of a "hasChild" role, (of another thing), it must be a person."

Properties of $\mathcal{AL}$

- $\mathcal{AL}$ has no "branching" in its tableaux (no union, or any kind of disjunction); so proofs in $\mathcal{AL}$ are linear.
- Note that all notions of RDFS can be expressed already in $\mathcal{AL}$:
  - C rdfs:subClassOf D: $C \sqsubseteq D$
  - p rdfs:subPropertyOf q: $p \sqsubseteq q$
  - p rdfs:domain C: $\exists p.\top \sqsubseteq C$
  - p rdfs:range C: $\top \sqsubseteq \forall p.C$
  - where C and D can be composite concept expressions over the $\mathcal{AL}$ constructors listed above.

SIMPLE DL LANGUAGES THAT INTRODUCE BRANCHING

- $\mathcal{U}$: "union"; e.g. Parent $\equiv$ Father $\sqcup$ Mother.
- $\mathcal{E}$: (unrestricted) existential quantification of the form $\exists R.C$ (recall that $\mathcal{AL}$ allows only $\exists R.\top$).
  HasSon $\equiv \exists$hasChild.Male , for persons who have at least one male child,
  GrandParent $\equiv \exists$hasChild.(hasChild.$\top$) for grandparents.

  Note: the FOL equivalent uses variables:
  hasSon$(x) \leftrightarrow \exists y(\text{hasChild}(x, y) \wedge \text{Male}(y))$,
  grandparent$(x) \leftrightarrow \exists y(\text{hasChild}(x, y) \wedge \exists x : \text{hasChild}(y, x))$.
- Exercise: show why unrestricted existential quantification $\exists R.C$ in contrast to $\exists R.\top$ leads to branching.
- A frequently used restriction of $\mathcal{AL}$ is called $\mathcal{FL}^-$ (for "Frame-Language"), which is obtained by disallowing negation completely (i.e., having only positive knowledge).

FAMILY OF DL LANGUAGES: $\mathcal{ALC}$ AND FURTHER EXTENSIONS

- $\mathcal{C}$: negation ("complement") of non-atomic concepts.
  Childless $\equiv$ Person $\sqcap \neg\exists$hasChild.$\top$ characterizes the set of persons who have no children (note: open-world semantics of negation!)

  Note: the FOL equivalent would be expressed via variables:
  $\forall x(\text{Childless}(x) \leftrightarrow (\text{Person}(x) \wedge \neg\exists y(\text{hasChild}(x, y))))$
- $\mathcal{U}$ and $\mathcal{E}$ can be expressed by $\mathcal{C}$.
  (by $A \cup B \equiv \neg(\neg A \wedge \neg B)$ and $\exists R.C \equiv \neg(\forall R.(\neg C))$).
- $\mathcal{ALC}$ is the "smallest" Description Logic that is closed wrt. the set operations.
- $\mathcal{N}$: (unqualified) cardinalities of roles ("number restrictions").
  $(\geq 3$ hasChild.$\top)$ for persons who have at least 3 children.
- $\mathcal{Q}$: qualified role restrictions:
  $(\leq 2$ hasChild.Male$)$
  $\mathcal{F}$: like $\mathcal{Q}$, but restricted to cardinalites 0, 1 and "arbitrary".

COMPLEXITY AND DECIDABILITY: OVERVIEW

- Logic $\mathcal{L}^2$, i.e., FOL with only two (reusable) variable symbols is decidable.
- Full FOL is undecidable.
- DLs: incremental, modular set of semantical notions.
- only part of FOL is required for concept reasoning.
- $\mathcal{ALC}$ can be *expressed* by FOL, but then, the inherent semantics is lost $\rightarrow$ full FOL reasoner required.
- Actually, $\mathcal{ALC}$ can be encoded in FOL by only using two variables $\rightarrow$ $\mathcal{ALC}$ is decidable.
- Consistency checking of $\mathcal{ALC}$-TBoxes and -ABoxes is PSPACE-complete (proof by reduction to *Propositional Dynamic Logic* which is in turn a special case of propositional multimodal logics).
  There are algorithms that are efficient in the average case.
- $\mathcal{ALCN}$ goes beyond $\mathcal{L}^2$ and PSPACE. Reduction to $\mathcal{C}^2$ (including "counting" quantifiers) yields decidability, but now in NEXPTIME. There are algorithms for $\mathcal{ALCN}$ and even $\mathcal{ALCQ}$ in PSPACE.

FURTHER EXTENSIONS

- Role hierarchy ($\mathcal{H}$; role subsumption and role equivalence, union/intersection of roles):
  hasSon $\sqsubseteq$ hasChild , hasChild $\equiv$ hasSon $\sqcup$ hasDaughter
- *Role Constructors* similar to regular expressions:
  concatenation (hasGrandchild $\equiv$ hasChild $\circ$ hasChild), transitive closure (hasDescendant $\equiv$ hasChild$^+$) (indicated by e.g. $\mathcal{H}_{reg}$ or $\mathcal{R}$), and inverse (isChildOf $\equiv$ hasChild$^-$) ($\mathcal{I}$).
- *Data types* (indicated by "(D)"), e.g. integers.
  Adult $\equiv$ Person $\sqcap \exists$age. ≥ 18.
- *Nominals* ($\mathcal{O}$) allow to use individuals from the ABox also in the TBox.
  Enumeration Concepts: BeNeLux $\equiv$ {Belgium, Netherlands, Luxemburg},
  HasValue-Concepts: GermanCity $\equiv \exists$inCountry.Germany.
- Role-Value-Maps:
  Equality Role-Value-Map: $(R_1 \equiv R_2)(x) \Leftrightarrow \forall y : R_1(x,y) \leftrightarrow R_2(x,y)$.
  Containment Role-Value-Map: $(R_1 \sqsubseteq R_2)(x) \equiv \forall y : R_1(x,y) \rightarrow R_2(x,y)$.
  (knows $\sqsubseteq$ likes) describes the set of people who like all people they know;
  i.e., (knows $\sqsubseteq$ likes)(john) denotes that John likes all people he knows.

FORMAL SEMANTICS OF EXPRESSIONS

- $I(\geq nR.C) = \{x \mid \#\{y \mid (x,y) \in I(R) \text{ and } y \in I(C)\} \geq n\}$,
- $I(\leq nR.C) = \{x \mid \#\{y \mid (x,y) \in I(R) \text{ and } y \in I(C)\} \leq n\}$,
- $I(nR.C) = \{x \mid \#\{y \mid (x,y) \in I(R) \text{ and } y \in I(C)\} = n\}$,
- $I(R \sqcup S) = I(R) \cup I(S)$, $I(R \sqcap S) = I(R) \cap I(S)$,
- $I(R \circ S) = \{(x,z) \mid \exists y : (x,y) \in I(R) \text{ and } (y,z) \in I(S)\}$,
- $I(R^-) = \{(y,x) \mid (x,y) \in I(R)\}$,
- $I(R^+) = (I(R))^+$.
- If nominals are used, $\mathcal{I}$ also assigns an element $I(x) \in \mathcal{D}$ to each nominal symbol x (similar to constant symbols in FOL). With this,
  $I(\{x_1, \ldots, x_n\}) = \{I(x_1), \ldots, I(x_n)\}$, and
  $I(R.y) = \{x \mid \{z \mid (x,z) \in I(R)\} = \{y\}\}$,
- $I(R_1 \equiv R_2) = \{x \mid \forall y : R_1(x,y) \leftrightarrow R_2(x,y)\}$,
  $I(R_1 \sqsubseteq R_2) = \{x \mid \forall y : R_1(x,y) \rightarrow R_2(x,y)\}$.

OVERVIEW: COMPLEXITY OF EXTENSIONS

- $\mathcal{ALC}_{reg}$, $\mathcal{ALCHIQ}_{\mathcal{R}^+}$, $\mathcal{ALCIO}$ are ExpTime-complete, $\mathcal{ALCHIQO}_{\mathcal{R}^+}$ is NExpTime-Complete.,
- Combining *composite* roles with cardinalities becomes undecidable (encoding in FOL requires 3 variables).
- Encoding of Role-Value Maps with composite roles in FOL is undecidable (encoding in FOL requires 3 variables; the logic loses the *tree model property*).
- $\mathcal{ALCQI}_{reg}$ with role-value maps restricted to boolean compositions of *basic* roles remains decidable. Decidability is also preserved when role-value-maps are restricted to functional roles.

DESCRIPTION LOGIC MODEL THEORY

The definition is the same as in FOL:

- an interpretation is a model of an ABox A if
  - for every atomic concept C and individual x such that $C(x) \in A$, $I(x) \in I(C)$, and
  - for every atomic role R and individuals x, y such that $R(x, y) \in A$, $(I(x), I(y)) \in I(R)$.
- note: the interpretation of the non-atomic concepts and roles is given as before,
- all axioms ϕ of the TBox are satisfied, i.e., $\mathcal{I} \models \phi$.

Based on this, DL entailment is also defined as before:

- a set Φ of formulas entails another formula Ψ (denoted by $\Phi \models \psi$), if $\mathcal{I}(\Psi) =$ true in all models $\mathcal{I}$ of Φ.

DECIDABILITY, COMPLEXITY, AND ALGORITHMS

Many DLs are decidable, but in high complexity classes.

- decidability is due to the fact that often *local* properties are considered, and the verification proceeds tree-like through the graph without connections between the branches.
- This locality does not hold for cardinalities over composite roles, and for role-value maps – these lead to undecidability.
- Reasoning algorithms for $\mathcal{ALC}$ and many extensions are based on tableau algorithms, some use model checking (finite models), others use tree automata.

Three types of Algorithms

- restricted (to polynomial languages) and complete
- expressive logics with complete, worst-case EXPTIME algorithms that solve realistic problems in "reasonable" time. (Fact, HermiT, Racer, Pellet)
- more expressive logics with incomplete reasoning.

EXAMPLE

- Given facts: Person $\equiv$ Male $\sqcup$ Female and Person(unknownPerson).
- Query ?-Male(X) yields an empty answer
- Query ?-Female(X) yields an empty answer
- Query ?-(Male $\sqcup$ Female)(X) yields unknownPerson as an answer
- for query answering, *all* models of the TBox+ABox are considered.
- in some models, the unknownPerson is Male, in the others it is female.
- in all models it is in (Male $\sqcup$ Female).

SUMMARY AND COMPARISON WITH FOL

Base Data (DL atomic concepts and atomic roles $\sim$ RDF)

- unary predicates (concepts/classes): Person(john),
- binary predicates (roles/properties): hasChild(john,alice)

Expressions

Concept/Role Expressions act as unary/binary predicates:

- $(\exists \text{ hasChild.Male})(\text{john})$, $(\text{Adult} \sqcap \text{Parent})(\text{john})$,
- $(\text{hasChild} \circ \text{hasChild})(\text{jack,alice})$, $(\text{neighbor}^*)(\text{portugal,germany})$

$\Rightarrow$ disjunction, conjunction and quantifiers *only* in the restricted contexts of expressions

$\Rightarrow$ implications *only* in the restricted contexts of TBox Axioms:

- $C_1 \sqsubseteq C_2$ $\quad \text{Parent} \sqsubseteq \text{Person}$
- $R_1 \sqsubseteq R_2$ $\quad \text{capital} \sqsubseteq \text{hasCity}$
- $C_1 \equiv C_2$ $\quad \text{Parent} \equiv \exists \text{hasChild}.\top$
- $R_1 \equiv R_2$ $\quad \text{neighbor} \equiv (\text{neighbor} \sqcup \text{neighbor}^-)$

$\Rightarrow$ ABox/TBox (= knowledge base) is a conjunctive set of atoms.

$\Rightarrow$ No formulas with $\wedge$, $\vee$, $\neg$, $\forall x$, $\exists x$!

9.2 OWL

- the OWL versions use certain DL semantics:
- Base: $\mathcal{ALC}_{\mathcal{R}^+}$: (i.e., with transitive roles). This logic is called $\mathcal{S}$ (reminiscent to its similarity to the modal logic S).
- roles can be ordered hierarchically (rdfs:subPropertyOf; $\mathcal{H}$).
- OWL Lite: $\mathcal{SHIF}(D)$, Reasoning in EXPTIME.
- OWL DL: $\mathcal{SHOIN}(D)$, decidable.
  Pellet (2007) implements $\mathcal{SHOIQ}(D)$. Decidability is in NEXPTIME (combined complexity wrt. TBox+ABox), but the actual complexity of a given task is constrained by the maximal used cardinality and use of nominals and inverses and behaves like the simpler classes.
  (Ian Horrocks and Ulrike Sattler: A Tableau Decision Procedure for SHOIQ(D); In IJCAI, 2005, pp. 448-453; available via `http://dblp.uni-trier.de`)
- OWL 2.0 towards $\mathcal{SROIQ}(D)$ and more datatypes ...

OWL Notions; OWL-DL vs. RDF/RDFS; Model vs. Graph

- OWL is defined based on (Description Logics) model theory,
- OWL ontologies can be represented by RDF graphs,
- Only *certain* RDF graphs are allowed OWL-DL ontologies: those, where individuals, (user-defined) properties, classes, and OWL's predefined properties etc. occur in a well-organized way.
  - user-defined properties are only allowed between individuals.
  - Relationships talking about classes and properties are restricted to those predefined in RDFS and OWL.
  - (user-defined *AnnotationProperties* (cf. Slide 420) can be used to associate names etc. with classes and properties, they are ignored by the reasoner, they "exist" only on the RDF (data) graph level)
- Reasoning works on the (Description Logic) model, the RDF graph is only a means to represent it.
  (recall: RDF/RDFS "reasoning" works on the RDF (data) graph level)

OWL VOCABULARIES

- An OWL-DL vocabulary $\mathcal{V}$ is a 7-tuple (= a sorted vocabulary) $\mathcal{V} = (\mathcal{V}_{cls}, \mathcal{V}_{objprop}, \mathcal{V}_{dtprop}, \mathcal{V}_{annprop}, \mathcal{V}_{indiv}, \mathcal{V}_{DT}, \mathcal{V}_{lit})$:
- $\mathcal{V}_{cls}$ is the set of URIs denoting class names,
  `<http://.../mondial/10/meta#Country>`
- $\mathcal{V}_{objprop}$ is the set of URIs denoting object property names,
  `<http://.../mondial/10/meta#capital>`
- $\mathcal{V}_{dtprop}$ is the set of URIs denoting datatype property names,
  `<http://.../mondial/10/meta#population>`
- ($\mathcal{V}_{annprop}$ is the set of URIs denoting annotation property names,)
- $\mathcal{V}_{indiv}$ is the set of URIs denoting individuals, `<http://.../mondial/10/countries/D>`
- $\mathcal{V}_{DT}$ is the set of URIs denoting datatype names,
  `<http://www.w3.org/2001/XMLSchema#int>`
- $\mathcal{V}_{lit}$ is the set of literals;
- the builtin notions (=URIs) from RDF, RDFS, OWL namespaces do not belong to the vocabulary of the ontology (they are only used for describing the ontology in RDF).

OWL INTERPRETATIONS

Since DL is a subset of FOL, the interpretation of an OWL-DL vocabulary can be given as a FOL interpretation

$$\mathcal{I} = (I_{indiv} \cup I_{cls} \cup I_{objprop} \cup I_{dtprop} \cup I_{annprop} \cup I_{DT} \ , \ \mathcal{U}_{obj} \cup \mathcal{U}_{DT})$$

where I interprets the vocabulary as

- I_{indiv} constant symbols (individuals),
- I_{cls}, I_{DT} unary predicates (classes and datatypes),
- $I_{objprop}$, I_{dtprop}, $I_{annprop}$ binary predicates (properties),

and the universe $\mathcal{U}$ is partitioned into

- an *object domain* $\mathcal{U}_{obj}$
- and a data domain $\mathcal{U}_{DT}$ (the literal values of the datatypes).

OWL INTERPRETATIONS

The interpretation I is as follows:

I_{indiv}: each individual $a \in \mathcal{V}_{indiv}$ to an object $I(a) \in \mathcal{U}_{obj}$,
(e.g., $I(\texttt{<http://.../mondial/10/countries/D>}) = \textit{germany}$)

I_{cls}: each class $C \in \mathcal{V}_{cls}$ to a set $I(C) \subseteq \mathcal{U}_{obj}$,
(e.g., $\textit{germany} \in I(\texttt{<http://.../mondial/10/meta\#Country>})$)

I_{DT}: each datatype $D \in \mathcal{V}_{DT}$ to a set $I(D) \subseteq \mathcal{U}_{DT}$,
(e.g., $I(\texttt{<http://www.w3.org/2001/XMLSchema\#int>}) = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$)

$I_{objprop}$: each object property $p \in \mathcal{V}_{objprop}$ to a binary relation $I(p) \subseteq \mathcal{U}_{obj} \times \mathcal{U}_{obj}$,
(e.g., $(\textit{germany}, \textit{berlin}) \in I(\texttt{<http://.../mondial/10/meta\#capital>})$)

I_{dtprop}: each datatype property $p \in \mathcal{V}_{dtprop}$ to a binary relation $I(p) \subseteq \mathcal{U}_{obj} \times \mathcal{U}_{D}$,
(e.g., $(\textit{germany}, 83536115) \in I(\texttt{<http://.../mondial/10/meta\#population>})$)

$I_{annprop}$: each annotation property $p \in \mathcal{V}_{annprop}$ to a binary relation $I(p) \subseteq \mathcal{U} \times \mathcal{U}$.

OWL Class Definitions and Axioms (Overview)

- owl:Class
- The properties of an owl:Class (including owl:Restriction) node describe the properties of that class.
  An owl:Class is required to satisfy the conjunction of all constraints (implicit: intersection) stated about it.
  These characterizations are roughly the same as discussed for DL class definitions:
  - Set-theory constructors: owl:unionOf, owl:intersectionOf, owl:complementOf ($\mathcal{ALC}$)
  - Enumeration constructor: owl:oneOf (enumeration of elements; $\mathcal{O}$)
  - Axioms rdfs:subClassOf, owl:equivalentClass,
  - Axiom owl:disjointWith (also expressible in $\mathcal{ALC}$: C disjoint with D is equivalent to $C \sqsubseteq \neg D$)

OWL NOTIONS (CONT'D)

OWL Restriction Classes (Overview)

- owl:Restriction is a subclass of owl:Class, allowing for specification of a constraint on *one* property.
- one property is restricted by an owl:onProperty specifier and a constraint on this property:
  - ($\mathcal{N}$, $\mathcal{Q}$, $\mathcal{F}$) owl:cardinality, owl:minCardinality or owl:maxCardinality,
  - owl:allValuesFrom ($\forall R.C$), owl:someValuesFrom ($\exists R.C$),
  - owl:hasValue ($\mathcal{O}$),
  - including datatype restrictions for the range (D)
- by defining intersections of owl:Restrictions, classes having multiple such constraints can be specified.

OWL Notions (Cont'd)

OWL Property Axioms (Overview)

- Distinction between owl:ObjectProperty and owl:DatatypeProperty
- from RDFS: rdfs:domain/rdfs:range assertions, rdfs:subPropertyOf
- Axiom owl:equivalentProperty
- Axioms: subclasses of rdf:Property:
  owl:TransitiveProperty, owl:SymmetricProperty, owl:FunctionalProperty,
  owl:InverseFunctionalProperty (see Slide 386)

OWL Individual Axioms (Overview)

- Individuals are modeled by unary classes
- owl:sameAs, owl:differentFrom, owl:AllDifferent($o_1,\ldots,o_n$).

First-Order Logic Equivalents

OWL : $x \in C$	DL Syntax	FOL
C	C	$C(x)$
intersectionOf(C_1, C_2)	$C_1 \sqcap \ldots \sqcap C_n$	$C_1(x) \wedge \ldots \wedge C_n(x)$
unionOf(C_1, C_2)	$C_1 \sqcup \ldots \sqcup C_n$	$C_1(x) \vee \ldots \vee C_n(x)$
complementOf(C_1)	$\neg C_1$	$\neg C_1(x)$
oneOf$(x_1, \ldots, x_n)$	$\{x_1\} \sqcup \ldots \sqcup \{x_n\}$	$x = x_1 \vee \ldots \vee x = x_n$

OWL : $x \in C$, Restriction on P	DL Syntax	FOL
someValuesFrom(C')	$\exists P.C'$	$\exists y : P(x, y) \wedge C'(y)$
allValuesFrom(C')	$\forall P.C'$	$\forall y : P(x, y) \rightarrow C'(y)$
hasValue(y)	$\exists P.\{y\}$	$P(x, y)$
maxCardinality(n)	$\leq n.P$	$\exists^{\leq n} y : P(x, y)$
minCardinality(n)	$\geq n.P$	$\exists^{\geq n} y : P(x, y)$
cardinality(n)	$n.P$	$\exists^{=n} y : P(x, y)$

First-Order Logic Equivalents (Cont'd)

OWL Class Axioms for C	DL Syntax	FOL
rdfs:subClassOf(C_1)	$C \sqsubseteq C_1$	$\forall x : C(x) \rightarrow C_1(x)$
equivalentClass(C_1)	$C \equiv C_1$	$\forall x : C(x) \leftrightarrow C_1(x)$
disjointWith(C_1)	$C \sqsubseteq \neg C_1$	$\forall x : C(x) \rightarrow \neg C_1(x)$

OWL Individual Axioms	DL Syntax	FOL
x_1 sameAs x_2	$\{x_1\} \equiv \{x_2\}$	$x_1 = x_2$
x_1 differentFrom x_2	$\{x_1\} \sqsubseteq \neg\{x_2\}$	$x_1 \neq x_2$
AllDifferent($x_1, \ldots, x_n$)	$\bigwedge_{i \neq j} \{x_i\} \sqsubseteq \neg\{x_j\}$	$\bigwedge_{i \neq j} x_i \neq x_j$

First-Order Logic Equivalents (Cont'd)

OWL Properties	DL Syntax	FOL
P	P	$P(x, y)$

OWL Property Axioms for P	DL Syntax	FOL
rdfs:range(C)	$\top \sqsubseteq \forall P.C$	$\forall x, y : P(x, y) \rightarrow C(y)$
rdfs:domain(C)	$C \sqsupseteq \exists P.\top$	$\forall x, y : P(x, y) \rightarrow C(x)$
subPropertyOf(P_2)	$P \sqsubseteq P_2$	$\forall x, y : P(x, y) \rightarrow P_2(x, y)$
equivalentProperty(P_2)	$P \equiv P_2$	$\forall x, y : P(x, y) \leftrightarrow P_2(x, y)$
inverseOf(P_2)	$P \equiv P_2^-$	$\forall x, y : P(x, y) \leftrightarrow P_2(y, x)$
TransitiveProperty	$P^+ \equiv P$	$\forall x, y, z : ((P(x, y) \wedge P(y, z)) \rightarrow P(x, z))$
		$\forall x, z : ((\exists y : P(x, y) \wedge P(y, z)) \rightarrow P(x, z))$
FunctionalProperty	$\top \sqsubseteq \leq 1 P.\top$	$\forall x, y_1, y_2 : P(x, y_1) \wedge P(x, y_2) \rightarrow y_1 = y_2$
InverseFunctionalProperty	$\top \sqsubseteq \leq 1 P^-.\top$	$\forall x, y_1, y_2 : P(y_1, x) \wedge P(y_2, x) \rightarrow y_1 = y_2$

SYNTACTICAL REPRESENTATION

- OWL specifications can be represented by graphs: OWL constructs have a straightforward representation as triples in RDF/XML and Turtle.
- there are several logic-based representations (e.g. *Manchester OWL Syntax*); TERP (which can be used with pellet) is a combination of Turtle and Manchester syntax.
- OWL in RDF/XML format: usage of class, property, and individual names:
  - as @rdf:about when used as identifier of a subject (owl:Class, rdf:Property and their subclasses),
  - as @rdf:resource as the object of a property.
- some constructs need auxiliary structures (collections):
  owl:unionOf, owl:intersectionOf, and owl:oneOf are based on Collections
  - representation in RDF/XML by rdf:parseType="Collection".
  - representation in Turtle by (x_1 x_2 ... x_n)
  - as RDF lists: rdf:List, rdf:first, rdf:rest

REQUIREMENT

- every entity in an OWL ontology must be explicitly typed (i.e., as a class, an object property, a datatype property, . . . , or an instance of some class).
  (for reasons of space this is not always done in the examples; in general, it may lead to incomplete results)

QUERYING OWL DATA

- queries are atomic and conjunctive DL queries against the underlying OWL-DL model.
- this model can still be seen as a graph:
  - many of the edges are those known from the basic RDF graph
  - some edges (and collections) are only there for encoding OWL stuff (describing owl:unionOf, owl:propertyChain etc.) – these should not be queried
- SPARQL-DL is a subset of SPARQL: not every SPARQL query pattern is allowed for use on an OWL ontology
  (but the reasonable ones are, so in practice this is not a problem.)
- the query language SPARQL-DL allows exactly such well-sorted patterns using the notions of OWL.

SOME TBOX-ONLY REASONING EXAMPLES ON SETS

Example: A Simple Paradox

```
@prefix : <foo://bla/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:Paradox owl:complementOf :Paradox.
```

[Filename: RDF/paradox.n3]

- without reasoner:

  jena -t -ol rdf/xml -if paradox.n3

  Outputs the same RDF facts in RDF/XML without checking consistency.

- with reasoner:

  jena -e -pellet -if paradox.n3

  reads the RDF file, creates a model (and checks consistency) and in this case reports that it is not consistent:
  "There is an anonymous individual which is forced to belong to class foo://bla/Paradox and its complement"

- Note: the reasoner invents an anonymous individual for checking consistency.
  The empty interpretation (with empty domain!) would be a model of $P \equiv \neg P$.

UNION AS $A \sqcup B \equiv \neg((\neg A) \sqcap (\neg B))$ (DE MORGAN'S RULE)

```
@prefix : <foo://bla/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:A rdf:type owl:Class.          :B rdf:type owl:Class.
:Union1 owl:equivalentClass [ owl:unionOf (:A :B) ].
:CompA owl:complementOf :A.     :CompB owl:complementOf :B.
:IntersectComps owl:equivalentClass [ owl:intersectionOf (:CompA :CompB)].
:Union2 owl:complementOf :IntersectComps.
:x rdf:type :A.                 :x rdf:type :B.
:y rdf:type :CompA. # a negative assertion y not in A would be better -> OWL 2
:y rdf:type :CompB.
```

[Filename: RDF/union.n3]

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix : <foo://bla/>
select ?X ?C ?D
from <file:union.n3>
where {{?X rdf:type ?C} UNION {:Union1 owl:equivalentClass ?D}}
```

[Filename: RDF/union.sparql]

EXAMPLE: UNION AND SUBCLASS

```
@prefix : <foo://bla/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:Male a owl:Class.      ## if these lines are missing,
:Female a owl:Class.    ## the reasoner complains
:Person owl:equivalentClass [ owl:unionOf (:Male :Female) ].
:EqToPerson owl:equivalentClass [ owl:unionOf (:Female :Male) ].
:unknownThing a [ owl:unionOf (:Female :Male) ].
```

[Filename: RDF/union-subclass.n3]

- print class tree (with jena -e -pellet -if union-subclass.n3):

```
owl:Thing
   bla:Person = bla:EqToPerson - (bla:unknownThing)
      bla:Female
      bla:Male
```

- Male and Female are derived to be subclasses of Person.
- Person and EqToPerson are equivalent classes.
- unknownThing is a member of Person and EqToPerson.

Example (Cont'd)

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla/>
select ?SC ?C ?T ?CC ?CD
from <file:union-subclass.n3>
where {{?SC rdfs:subClassOf ?C} UNION
       {:unknownPerson rdf:type ?T} UNION
       {?CC owl:equivalentClass ?CD}}
```

[Filename: RDF/union-subclass.sparql]

- Note: OWLizations of DL class expressions are always handled as blank nodes, and used with "owl:equivalentClass", "rdf:subClassOf", "rdfs:domain", "rdfs:range" or "a".

Aside: the same in RDF/XML (usage of rdf:parseType="Collection")

```
<?xml version="1.0"?>
<rdf:RDF xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:f="foo://bla/"
    xml:base="foo://bla/">
 <owl:Class rdf:about="Person">
  <owl:equivalentClass>
   <owl:Class>
    <owl:unionOf rdf:parseType="Collection">
     <owl:Class rdf:about="Male"/>
     <owl:Class rdf:about="Female"/>
    </owl:unionOf>
   </owl:Class>
  </owl:equivalentClass>
 </owl:Class>
 <owl:Class rdf:about="EqToPerson">
  <owl:equivalentClass>
   <owl:Class>
    <owl:unionOf rdf:parseType="Collection">
     <owl:Class rdf:about="Female"/>
     <owl:Class rdf:about="Male"/>
    </owl:unionOf>
   </owl:Class>
  </owl:equivalentClass>
 </owl:Class>
 <f:Person rdf:about="unknownPerson"/>
</rdf:RDF>
```

[Filename: RDF/union-subclass.rdf]

EXERCISE

Consider

```
<owl:Class rdf:about="C1">
 <owl:equivalentClass>
  <owl:Class>
   <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="A"/>
    <owl:Class rdf:about="B"/>
   </owl:intersectionOf>
  </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

and

```
<owl:Class rdf:about="C2">
 <rdfs:subClassOf rdf:resource="A"/>
 <rdfs:subClassOf rdf:resource="B"/>
</owl:Class>
```

- give mathematical characterizations of both cases.
- discuss whether both fragments are equivalent or not.

DISCUSSION

- Two classes are *equivalent* (wrt. the knowledge base) if they have the same interpretation in every *model* of the KB.
- C_1 is characterized to be the intersection of classes A and B.
- for C_2, it is asserted that C_2 is a subset of A and that it is a subset of B.
- Thus there can be some c that is in A, B, C_1, but not in C_2.
- Thus, C_1 and C_2 are not equivalent.
- C_1 is a definition, the statements about C_2 are just two constraints (C_2 might be empty); but it can be derived that it must be a subclass of C_1.

DISCUSSION: FORMAL NOTATION

The DL equivalent to the knowledge base (TBox) is

$$\mathcal{T} = \{C_1 \equiv (A \sqcap B)\ , \quad C_2 \sqsubseteq A\ , \quad C_2 \sqsubseteq B\}$$

The First-Order Logic equivalent is

$$\mathcal{KB} = \{\forall x : A(x) \wedge B(x) \leftrightarrow C_1(x)\ , \quad \forall x : C_2(x) \rightarrow A(x) \wedge B(x)\}$$

Thus, $\mathcal{KB} \models \forall x : C_2(x) \rightarrow A(x) \wedge B(x)$.

Or, in DL: $\mathcal{T} \models C_2 \sqsubseteq C_1$.

On the other hand, $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ with $\mathcal{D} = \{c\}$ and

$$\mathcal{I}(A) = \{c\},\ \ \mathcal{I}(B) = \{c\},\ \ \mathcal{I}(C_1) = \{c\},\ \ \mathcal{I}(C_2) = \emptyset$$

is a model of $\mathcal{KB}$ (wrt. first-order logic) and $\mathcal{T}$ (wrt. DL) that shows that C_1 and C_2 are not equivalent.

SUBCLASSES OF PROPERTIES

Triple syntax: *some property* rdf:type *a specific type of property*

According to their ranges

- owl:ObjectProperty – subclass of rdf:Property; object-valued (i.e. rdfs:range must be an Object class)
- owl:DatatypeProperty – subclass of rdf:Property; datatype-valued (i.e. its rdfs:range must be an rdfs:Datatype)

⇒ OWL ontologies require each property to be typed in such a way!
(for reasons of space sometimes omitted in examples)

According to their Cardinality

- specifying n:1 or 1:n cardinality:
owl:FunctionalProperty, owl:InverseFunctionalProperty

⇒ useful for deriving that objects must be different from each other.

According to their Properties

- owl:TransitiveProperty, owl:SymmetricProperty see later ...

FUNCTIONAL CARDINALITY SPECIFICATION

property rdf:type owl:FunctionalProperty

- not a constraint, but
- if such a property results in two things ... these things are inferred to be the same.

```
@prefix : <foo://bla/meta#>.
@prefix persons: <foo://bla/persons/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
  :world :has_pope persons:jorgebergoglio .
  :world :has_pope [ :name "Franziskus" ] .
  :has_pope rdf:type owl:FunctionalProperty.
```

[Filename: RDF/popes.n3]

```
prefix : <foo://bla/meta#>
prefix persons: <foo://bla/persons/>
select ?N  from <file:popes.n3>
where { persons:jorgebergoglio :name ?N }
```

[Filename: RDF/popes.sparql]

OWL:RESTRICTION – EXAMPLE

- owl:Restriction for $\exists p.C$ and $\forall p.C$. (cf. earlier examples)
- Definition of "Parent" as $\mathsf{Parent} \equiv \mathsf{Person} \sqcap \exists \mathsf{hasChild}.\top$ (can be used for conclusions in both directions),
- Range axiom as constraint: $\mathsf{Parent} \sqsubseteq \forall \mathsf{hasChild}.\mathsf{Person}$ (use only in the "$\Rightarrow$" direction)

```
@prefix : <foo://bla/meta#>.
@prefix p: <foo://bla/persons/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:Parent owl:equivalentClass
  [ owl:intersectionOf ( :Person
                         [ a owl:Restriction;
                             owl:onProperty :hasChild;  owl:minCardinality 1 ] ) ] .
:Parent rdfs:subClassOf [ a owl:Restriction;
                             owl:onProperty :hasChild;  owl:allValuesFrom :Person ] .
p:john a :Person; :hasChild p:alice .
p:sue a :Parent .
```

[Filename: RDF/restriction.n3]

owl:Restriction – Example (cont'd)

```
prefix : <foo://bla/meta#>
select ?X ?CC ?Y ?C
from <file:restriction.n3>
where {{?X a :Person; a ?CC} union {?Y :hasChild ?C}}
```

[File: RDF/restriction.sparql]

- How to check whether it knows that Sue has a child?
  - ... only *implicitly* known resources are never contained in SPARQL answers (impedance mismatch between SPARQL and DL).
  - they are only known *inside* the reasoner.
  - for looking inside the reasoner's "private" knowledge, appropriate auxiliary classes have to be defined in the OWL ontology which are then queried by SPARQL (as in many later examples)

- note also the separation of the domain into notions (<foo://bla/meta#>) and instances (<foo://bla/persons/>).
  This will not be cleanly done in the subsequent examples because it costs space.

Aside: owl:Restriction as RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="foo://bla/meta#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xml:base="foo://bla/persons/">
 <owl:Class rdf:about="Parent">
  <owl:equivalentClass>
   <owl:Class>
    <owl:intersectionOf rdf:parseType="Collection">
     <owl:Class rdf:about="Person"/>
     <owl:Restriction>
      <owl:onProperty rdf:resource="hasChild"/>
      <owl:minCardinality>1</owl:minCardinality>
     </owl:Restriction>
    </owl:intersectionOf>
   </owl:Class>
  </owl:equivalentClass>
 </owl:Class>
 <Person rdf:about="john">
   <hasChild><Person rdf:about="alice"/></hasChild>
 </Person>
</rdf:RDF>
```

[Filename: RDF/restriction.rdf]

RESTRICTIONS (AND OTHER CLASS SPECIFICATIONS) AS SEPARATE BLANK NODES

Consider the following (bad) specification:

```
@prefix : <foo://bla/meta#>.   @prefix p: <foo://bla/persons/>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:BadIdea a owl:Class; a owl:Restriction;
          owl:onProperty :hasChild; owl:minCardinality 1.
p:john :hasChild p:alice.
:cl a owl:Class.                          [Filename: RDF/restrictionWrong.n3]
```

This is not allowed in OWL-DL.

- Note [13.6.2019]: specifications of that form, not using a blank node, are ignored by jena3.10/pellet (the class BadIdea does not exist):

  jena -e -if restriction.n3 -if restrictionWrong.n3

Correct specification:

```
:goodIdea owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].
```

Why? ... there are many reasons, for one of them see next slide.

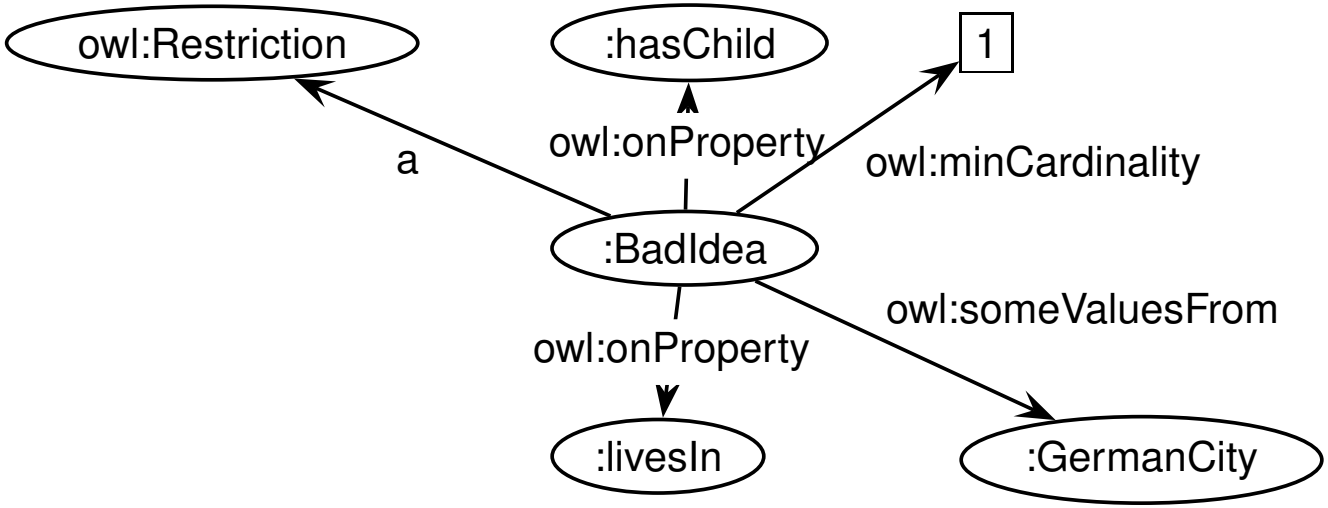
Restrictions Only as Blank Nodes (Cont'd)

A class with two such specifications:

```
@prefix : <foo://bla/meta#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:BadIdea a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1 .
:BadIdea a owl:Restriction; owl:onProperty :livesIn; owl:someValuesFrom :GermanCity.
```

[Filename: RDF/badIdea.n3]

- call jena -t -pellet -if badIdea.n3:



The two restriction specifications are messed up.

Restrictions Only as Blank Nodes (Cont'd)

- Thus specify each Restriction specification with a separate blank node:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/meta#>.
:TwoRestrictions owl:equivalentClass
 [ owl:intersectionOf
   ( [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1]
     [ a owl:Restriction; owl:onProperty :livesIn; owl:someValuesFrom :GermanCity] ) ].
```

[Filename: RDF/twoRestrictions.n3]

The DL equivalent: TwoRestrictions $\equiv$ ($\exists$ hasChild.$\top$) $\sqcap$ ($\exists$ livesIn.GermanCity)

Another reason:

```
:BadSpecOfParent a owl:Restriction;
   owl:onProperty :hasChild; owl:minCardinality 1;
   rdfs:subClassOf :Person.
```

... mixes the *definition* of the Restriction with an assertive axiom:
BSOP $\equiv \exists \geq 1$ hasChild.$\top \wedge$ ABDE $\sqsubseteq$ Person
(This expression probably does not meet the original intention – is *derives* that anything that has a child is made an instance of class "Person"; cf. Slide 383)

MULTIPLE RESTRICTIONS ON A PROPERTY

- "All persons that have at least two children, and one of them is male"
- first: a straightforward wrong attempt

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla/meta#>.   @prefix p: <foo://bla/persons/>.
### Test: multiple restrictions: the owl:someValuesFrom-condition is then ignored
:HasTwoChildrenOneMale owl:intersectionOf (:Person
   [ a owl:Restriction; owl:onProperty :hasChild;
     owl:someValuesFrom :Male; owl:minCardinality 2]).
:name a owl:FunctionalProperty.
:Male rdfs:subClassOf :Person; owl:disjointWith :Female.
:Female rdfs:subClassOf :Person.
:kate a :Female; :name "Kate"; :hasChild :john.
p:john a :Male; :name "John";
  :hasChild [a :Female; :name "Alice"], [a :Male; :name "Bob"].
p:sue a :Female; :name "Sue";
  :hasChild [a :Female; :name "Anne"], [a :Female; :name "Barbara"].
```

[Filename: RDF/restriction-double.n3]

```
prefix : <foo://bla/meta#>
select ?X
from <file:restriction-double.n3>
where {?X a :HasTwoChildrenOneMale}
```

[Filename: RDF/restriction-double.sparql]

- The the owl:someValuesFrom-condition is ignored in this case (Result: John and Sue).

Multiple Restrictions on a Property

- "All persons that have at least two children, and one of them is male"
- to expressed as an *intersection* of two separate restrictions:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla/meta#>.      @prefix p: <foo://bl
:HasTwoChildrenOneMale owl:equivalentClass
 [ owl:intersectionOf (:Person
   [ a owl:Restriction; owl:onProperty :hasChild; owl:someValuesFrom :Male]
   [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 2] ) ].
:name a owl:FunctionalProperty.
:Male rdfs:subClassOf :Person; owl:disjointWith :Female.
:Female rdfs:subClassOf :Person.
p:kate a :Female; :name "Kate"; :hasChild p:john.
p:john a :Male; :name "John";
  :hasChild [a :Female; :name "Alice"], [a :Male; :name "Bob"].
p:sue a :Female; :name "Sue";
  :hasChild [a :Female; :name "Anne"], [a :Female; :name "Barbara"].
```

[Filename: RDF/intersect-restrictions.n3]

```
prefix : <foo://bla/meta#>
select ?X
from <file:intersect-restrictions.n3>
where {?X a :HasTwoChildrenOneMale}
```

[Filename: RDF/intersect-restrictions.sparql]

- Note: this is different from Qualified Range Restrictions such as "All persons that have at least two male children" – see Slide 461.

Use of a Derived Class

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla/meta#>.      @prefix p: <foo://bla/persons/>.
p:kate :name "Kate"; :hasChild p:john.
p:john :name "John"; :hasChild p:alice.
p:alice :name "Alice".
:Parent a owl:Class; owl:equivalentClass
 [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].
:Grandparent owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasChild; owl:someValuesFrom :Parent].
```

[Filename: RDF/grandparent.n3]

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix : <foo://bla/meta#>
select ?G ?P ?GP
from <file:grandparent.n3>
where {{?G a :Parent} UNION
       {?GP a :Grandparent} UNION
       {:Grandparent rdfs:subClassOf :Parent}}
```

[Filename: RDF/grandparent.sparql]

NON-EXISTENCE OF PROPERTY FILLERS (POSSIBLE SYNTAXES)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/meta#>.    @prefix p: <foo://bla/persons/>.
:ChildlessA owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :hasChild; owl:maxCardinality 0]).
:ChildlessB owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :hasChild; owl:allValuesFrom owl:Nothing]).
:ParentA owl:intersectionOf (:Person [owl:complementOf :ChildlessA]).          ### (*)
:ParentB owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1]).
:name a owl:FunctionalProperty.
p:john a :Person; :name "John"; :hasChild p:alice, p:bob.
p:sue a :ParentA; :name "Sue".
p:george a :Person; a :ChildlessA; :name "George".
```

[Filename: RDF/parents-childless.n3]

- export class tree: ChildlessA and ChildlessB are equivalent,
- ParentA and ParentB are also equivalent
- note: due to the Open World Assumption, only George is definitely known to be childless.
- Persons where parenthood is not known (Alice, Bob) are neither in Childless nor in Parent!
  Note: (*) states “Parent” vs. “Childless” as a disjoint, total partition of “Person”, but it is not *known* to which partition Alice and Bob belong. Both would be possible.

NON-EXISTENCE OF PROPERTY FILLERS – OPEN WORLD VS. CLOSED WORLD

- basically the same, Parent and Childless as classes, more persons,
- the focus is now on the different explicit and implicit knowledge about them:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/meta#>.    @prefix p: <foo://bla/persons/>.
:Childless owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :hasChild; owl:maxCardinality 0]).
:Parent owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1]).
:name a owl:FunctionalProperty.
p:kate a :Person; :name "Kate"; :hasChild p:john, p:sue.
p:john a :Person; :name "John"; :hasChild p:alice, p:bob.
p:alice a :Person; :name "Alice".
p:bob a :Person; :name "Bob".
p:sue a :Parent; :name "Sue".
p:george a :Person; a :Childless; :name "George".
```

[Filename: RDF/childless.n3]

```
prefix : <foo://bla/meta#>
select ?CL ?NCL ?P ?NP  ?X ?Y ?NHC from <file:childless.n3>
where {      {?CL a :Childless}
       union {?NCL a :Person FILTER NOT EXISTS { ?NCL a :Childless}}
       union {?P a :Parent}
       union {?NP a :Person FILTER NOT EXISTS { ?NP a :Parent}}
       union {?X :hasChild ?Y}
       union {?NHC a :Person FILTER NOT EXISTS {?NHC :hasChild ?Z}}}
```

[Filename: RDF/childless.sparql]

DL (and OWL) – everything that is done *inside the reasoner*: open world – monotonic, SPARQL: closed-world – non-monotonic:

- ?CL: only George is known to be Childless.
- ?NCL: Closed-World-Complement of ?C – all persons where it cannot be proven that they are childless – "definitely not childless or maybe not childless" – "where it is consistent to assume that they are not childless" – non-monotonic (all except George).
- Parents ?P: Sue, Kate, John;
- ?NP: Closed-World-Complement of ?P – ("consistent to be non-parents" – George, Alice, Bob)
- ?X, ?Y: only explicitly known parents/children (Sue's children not mentioned).
- ?NHC: George, Alice, Bob and Sue(!) – no children of them are *explicitly known*.

INVERSE PROPERTIES

- *owl:ObjectProperty* owl:inverseOf *owl:ObjectProperty*
- owl:DatatypeProperties cannot have an inverse
  (this would define properties of objects, cf. next slide)

```
@prefix : <foo://bla/meta#>.
@prefix p: <foo://bla/persons/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:descendant rdf:type owl:TransitiveProperty.
:hasChild rdfs:subPropertyOf :descendant.
:hasChild owl:inverseOf :hasParent.
p:john :hasChild p:alice, p:bob.
p:john :hasParent p:kate .
```

[Filename: RDF/inverse.n3]

```
prefix : <foo://bla/meta#>
select ?X ?Y
from <file:inverse.n3>
where {?X :descendant ?Y}
```

[Filename: RDF/inverse.sparql]

No Inverses of owl:DatatypeProperties!

- an owl:DatatypeProperty must not have an inverse:
- ":john :age 35" would imply "35 :ageOf :john" which would mean that a literal has a property, which is not allowed.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <foo://bla/meta#> .  @prefix p: <foo://bla/persons/> .
# p:john :name "John"; :age 35;
#        :hasChild [:name "Alice"], [:name "Bob"; :age 8].
:age a owl:DatatypeProperty.
:hasChild a owl:ObjectProperty.
:parent owl:inverseOf :hasChild.
:ageOf owl:inverseOf :age.
```

[Filename: RDF/inverseDTProp.n3]

```
jena -e -pellet -if inverseDTProp.n3
WARN [main] (OWLLoader.java:352) - Unsupported axiom:
Ignoring inverseOf axiom between foo://bla/meta#ageOf (ObjectProperty)
  and foo://bla/meta#age (DatatypeProperty)
```

SPECIFICATION OF INVERSE FUNCTIONAL PROPERTIES

- Mathematics: a mapping m is inverse-functional if the inverse of m is functional:
  $x\ p\ y$ is inverse-functional, if for every y, there is at most one x such that xpy holds.
- Example:
  - hasCarCode is functional: every country has one car code,
  - hasCarCode is also inverse functional: every car code uniquely identifies a country.
- OWL:

  `:m-inverse owl:inverseOf :m .`

  `:m-inverse a owl:FunctionalProperty .`

  not allowed for e.g. `mon:carCode a owl:DatatypeProperty`:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:carCode a owl:DatatypeProperty; rdfs:domain :Country;
  owl:inverseOf :isCarCodeOf.
# :Germany :carCode "D".                    [Filename: RDF/noinverse.n3]
```

- the statement is rejected.

OWL:INVERSEFUNCTIONALPROPERTY

- such cases are described with owl:InverseFunctionalProperty
- a property P is an owl:InverseFunctionalProperty if
  $\forall x, y_1, y_2 : P(y_1, x) \wedge P(y_2, x) \rightarrow y_1 = y_2$ holds

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:bla#>.
:carCode rdfs:domain :Country; a owl:DatatypeProperty;
  a owl:FunctionalProperty; a owl:InverseFunctionalProperty.
:name a owl:DatatypeProperty; a owl:FunctionalProperty.
:Germany :carCode "D"; :name "Germany".
:DominicanRepublic :carCode "D"; :name "Dominican Republic".
```

[Filename: RDF/invfunctional.n3]

- the fragment is detected to be inconsistent.

Named and Unnamed Resources

(from the DL reasoner's perspective)

Named Resources

- resources with explicit global URIs
  <http://www.semwebtech.org/mondial/10/country/D>
  <foo://bla/bob>
- resources with local IDs/named blank nodes
- unnamed blank nodes

Unnamed (implicit) Resources

- things that exist only implicitly:
  John's child in

```
:Parent a owl:Class; owl:equivalentClass
   [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].
:john a :Parent.
```

- such implicit resources can even have properties (see next slides).

Implicit Resources

- “every person has a father who is a person” and “john is a person”.
- the *standard model* is *infinite*:
  john, john’s father, john’s father’s father, ...
- pure RDF graphs are always finite,
- only with OWL axioms, one can specify such infinite models,

$\Rightarrow$ they have only finitely many *locally to path length* n different nodes,

- the reasoner can detect the necessary n (“blocking”, cf. Slides 518 ff) and create “typical” different structures.

Aside: “standard model” vs “nonstandard model”

- the term “standard model” is not only “what we understand (in this case)”, but is a notion of mathematical theory which –roughly– means “the simplest model of a specification”
- nonstandard models of the above are those where there is a cycle in the ancestors relation.
  (as the length of the cycle is arbitrary, this would not make it easier for the reasoner - there is only the possibility to have an owl:sameAs somewhere)

Implicit Resources

```
@prefix : <foo://bla#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:Person owl:equivalentClass [a owl:Restriction;
  owl:onProperty :father; owl:someValuesFrom :Person].
:bob :name "Bob";  a :Person; :father :john.
:john :name "John";  a :Person.
```

[Filename: RDF/fathers-and-forefathers.n3]

```
prefix : <foo://bla#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X ?F ?C
from <file:fathers-and-forefathers.n3>
where {{ ?X :father ?F } UNION { ?C a :Person }}
```

[Filename: RDF/fathers-and-forefathers.sparql]

- Reasoner: works on the model, including blocking, i.e. *modulo equivalence up to paths of length* n.
- SPARQL (and SWRL) rules: works on the graph – without the unnamed/implicit resorces.

9.3 RDF Graph vs. OWL Model; SPARQL vs. Reasoning

- SPARQL is an RDF (graph) query language
- OWL talks about models
- the RDF graph contains triples
  - ABox statements
  - partially also TBox – DL concepts are represented by several triples *that do not necessarily belong to the graph*
- The reasoner works in terms of DL concepts
  - when parsing input (RDF) files (or when activating the reasoner the first time), the DL concepts and assertions must be extracted ans communicated to the reasoner
- some only existentially known skolem objects exist only inside the reasoner
- (see Slide 572 for a sketch of the Jena architecture)

Queries against Metadata - Consequences (Overview)

$\Rightarrow$ SPARQL queries are answered against the graph of triples

- Some OWL notions are directly represented by triples, such as c a owl:Class.
- Some others are directly supported by special handling in the reasoners, e.g., c rdfs:subClassOf d and c owl:equivalentClass d.
- some others are only "answered" when given explicitly in the RDF input! The results then do not incorporate further results that could be found by reasoning!
- OWL notions in the input are often not contained as triples, but are only translated into DL atoms for the reasoner. (e.g. owl:Restriction definitions)
- Most OWL notions in queries are not "understood" as OWL, but only matched.

Queries against Resources

- SPARQL answers are only concerned with the RDF graph, not with existential things that are only known in the reasoner's model.

METADATA/ONTOLOGY LEVEL QUERYING

- SPARQL is defined by *matching* the underlying RDF graph.
- OWL triples are not always part of the RDF graph (they are intended to be translated into DL definitions in the reasoner)
- for traditional DL notions like

```
?C a owl:Class
?C a rdfs:subClassOf ?D
?C owl:equivalentClass ?D
?C owl:disjointWith ?D
```

  SPARQL implementations support to translate these internally into DL queries against the reasoner.
- SPARQL-DL (Sirin, Parsia OWLED 2007 [members of the Pellet team]) is a proposal that allows certain further OWL built-ins to be queried.

Test: Querying Metadata

Example: querying metadata (i) as triples, and (ii) by reasoning.

- the triples describing owl:Restrictions from the input are also in the graph and can be queried; nevertheless, answers are incomplete:

(intersect-restrictions.n3: Slide 395, cats-and-dogs.n3: Slide 461)

... see next slide

```
prefix : <foo://bla#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?NC ?NC2 ?X ?P ?S ?A ?M ?C ?MQ ?Y ?Y2
from <file:intersect-restrictions.n3>
from <file:cats-and-dogs.n3>
where { ?X a owl:Restriction
        optional {?X owl:onProperty ?P}
        optional {?NC owl:equivalentClass ?X . filter(isURI(?NC))
                  optional {?NC rdfs:subClassOf ?Y . ?Y a owl:Restriction}}
        optional {?X owl:equivalentClass ?NC2 . filter(isURI(?NC2))}  ### see notes below!
        optional {?X rdfs:subClassOf ?Y2 . ?Y2 a owl:Restriction}
        optional {?X owl:onClass ?C}
        optional {?X owl:someValuesFrom ?S}
        optional {?X owl:allValuesFrom ?A}
        optional {?X owl:minCardinality ?M}
        optional {?X owl:minQualifiedCardinality ?MQ}}
```

[Filename: RDF/metadata-query.sparql]

- NC2 is never bound (although just symmetric to NC!)
- NC=HasTwoDogs subClassOf Y=b1 is found
- X subClassOf Y2 is never bound(!)

Ontology Level Querying - a practical example

Consider again the "Childless" ontology from Slide 398.
Check that $\text{Childless} \sqcap \text{Parent} = \emptyset$ and $\text{Person} \equiv \text{Childless} \sqcup \text{Parent}$ (Partitioning)

- Allowed: (single line empty bindings result means true)

```
prefix : <foo://bla/meta#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?X from <file:childless.n3>
where { :Childless owl:disjointWith :Parent }
```

[Filename: RDF/childless1.sparql]

- Not allowed: complex class expression in the query (empty result since it tries a plain match with the RDF data)

```
prefix : <foo://bla/meta#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?X from <file:childless.n3>
where { :Person owl:equivalentClass [ owl:unionOf (:Childless :Parent) ] }
```

[Filename: RDF/childless2.sparql]

NOT ALLOWED

- instead: add auxiliary class definition to the TBox and export class tree with `jena -e -if childless.n3 childless3.n3` :

```
@prefix : <foo://bla/meta#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:UnionCLP owl:equivalentClass [ owl:unionOf (:Childless :Parent) ] .
```

[Filename: RDF/childless3.n3]

NOT REASONED: OWL:FUNCTIONALPROPERTY

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:bla#>.
:q a owl:FunctionalProperty.
:p a owl:ObjectProperty; rdfs:domain :D.
:D owl:equivalentClass [ a owl:Restriction; owl:onProperty :p;
                                            owl:maxCardinality 1 ].
# :x :p :a, :b.    :a owl:differentFrom :b.
```

[Filename:RDF/functional.n3]

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo:bla#>
select ?P
from <file:functional.n3>
where { ?P a owl:FunctionalProperty }
```

[Filename:RDF/functional.sparql]

- tries just to match plain { ?P a owl:FunctionalProperty } triples in the RDF graph. Returns only q.
- does not *answer* that property q is in fact also functional, although the reasoner knows it.

NOT ALLOWED: COMPLEX TERMS IN SPARQL QUERIES

- example: all cities that are a capital
- works well with pellet alone (June 2017); not allowed with Jena

```
pellet query -query-file countrycaps.sparql \
       mondial-europe.n3 mondial-meta.n3 countrycaps.n3
```

- note: if the answer is empty, check that the mondial-namespace in the used mondial-meta.n3 is correct.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://www.semwebtech.org/mondial/10/meta#> .
:CountryCapital owl:intersectionOf
  (:City [a owl:Restriction; owl:onProperty :isCapitalOf;
          owl:someValuesFrom :Country]).
```

[Filename: RDF/countrycaps.n3]

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?N1 ?N2
where {{?X a :CountryCapital; :name ?N1} union
       {?Y a [a owl:Restriction; owl:onProperty :isCapitalOf;
              owl:someValuesFrom :Country]; :name ?N2}}
```

[Filename:RDF/countrycaps.sparql]

Not Allowed: Complex Terms in SPARQL Queries (Cont'd)

- all organizations whose headquarter city is a capital:
- neither allowed by pellet nor by jena+pellet (June 2017; worked with pellet alone in 2013)

```
pellet query -query-file organizations-query2.sparql \
  mondial-europe.n3 mondial-meta.n3
```

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?A ?H
where {?X a [ owl:intersectionOf
            (:Organization [a owl:Restriction; owl:onProperty :hasHeadq;
                            owl:someValuesFrom
                              [ a owl:Restriction; owl:onProperty :isCapitalOf;
                                owl:someValuesFrom :Country ] ] ) ];
       :abbrev ?A; :hasHeadq ?C . ?C :name ?H . }
```

[Filename:RDF/organizations-query2.sparql]

How to do it: Sets of Answers to Queries as Ad-hoc Concepts

- The result concept (and maybe others) must be added to the ontology.
- Example: all organizations whose headquarter city is a capital:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://www.semwebtech.org/mondial/10/meta#> .
:CountryCapital owl:equivalentClass
  [ owl:intersectionOf
    (:City [a owl:Restriction; owl:onProperty :isCapitalOf;
            owl:someValuesFrom :Country])].
<bla:Result>  owl:equivalentClass [ owl:intersectionOf
  (:Organization [a owl:Restriction; owl:onProperty :hasHeadq;
    owl:someValuesFrom :CountryCapital])] .
```

[Filename: RDF/organizations-query.n3]

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?A ?N
from <file:organizations-query.n3>
from <file:mondial-europe.n3>
from <file:mondial-meta.n3>
where {?X a <bla:Result> . ?X :abbrev ?A . ?X :hasHeadq ?C . ?C :name ?N}
```

[Filename:RDF/organizations-query.sparql]

SPARQL ON THE GRAPH: IMPLICITLY KNOWN RESOURCES

- SPARQL does not return any answer related with nodes (=resources) that are only implicitly known (=non-named resources)

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:ParentOf12YOChild  owl:equivalentClass [a owl:Restriction;
  owl:onProperty :hasChild; owl:someValuesFrom :12YOPerson].
:12YOPerson owl:equivalentClass [a owl:Restriction;
  owl:onProperty :age; owl:hasValue 12].
[ :name "John"; :age 35;  a :ParentOf12YOChild;
   :hasChild [:name "Alice"; :age 10], [:name "Bob"; :age 8]].
:age rdf:type owl:FunctionalProperty.
# :12YOPerson owl:equivalentClass owl:Nothing.

:TwoChildrenParent  owl:equivalentClass [a owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 2].
:ThreeChildrenParent  owl:equivalentClass [a owl:Restriction;
  owl:onProperty :hasChild; owl:minCardinality 3].
```

[Filename: RDF/john-three-children-impl.n3]

SPARQL and Non-Named Resources (Cont'd)

- implicit resources exist only on the reasoning level,
- not considered by SPARQL queries:

```
prefix : <foo://bla#>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
select ?X ?C ?A ?T
from <file:john-three-children-impl.n3>
where {{ ?X :name "John" . ?X a ?C }
       UNION {?X :age ?A} UNION {?T a :12YOPerson}}
```

[Filename: RDF/john-three-children-impl.sparql]

- John is a ThreeChildrenParent,
- no person known who is 12 years old
- adding `:12YOPerson owl:equivalentClass owl:Nothing` makes it inconsistent.
- implicity known things are also not considered for the OWL construct owl:hasKey (cf. Slides 421 ff.; owl:hasKey is motivated by the relational database model, and not by Description Logic) and for SWRL rules (cf. Slides 529 ff).

9.4 Miscellaneous and Borderline Things in OWL

- Up to now: typical, “basic” OWL-DL
  - “natural” small toy ontologies
- issues that break the OWL individuals–classes–properties design:
  - Ontology metadata: documentation – for the user, not for the reasoner
  - Ontology metadata: facts that are not acceptable for the reasoner
  - Data reification: data about data
- issues from classical data management:
  - (passive) integrity constraints: keys.
  - collections and lists: RDF container concepts; sometimes used for *expressing* OWL stuff via RDF triples
  - logical rules? – cf. Slides 529 ff. (SWRL) and 603 ff. (Jena Rule-Based Reasoner)

9.4.1 Annotation Properties

- in OWL, user-defined properties are only allowed between instances; for classes and properties only the predefined RDFS and OWL properties are allowed:
  - rdfs:subClassOf, rdfs:SubPropertyOf, rdfs:domain, rdfs:range, rdf:type, that still belong to the RDF graph,
  - owl:onProperty, owl:someValuesFrom that exist only in the N3 serialization of the DL-based OWL vocabulary

$\Rightarrow$ owl:AnnotationProperties can be used to associate names etc. with classes and properties, they are ignored by the reasoner, they "exist" only on the RDF (data) graph level (i.e., can be queried by SPARQL):

- predefined annotation properties: rdfs:label (assigning "names" to classes and properties), rdfs:comment, rdfs:seeAlso, rdfs:isDefinedBy,
- commonly used annotation properties for provenance and metadata: e.g., Dublin Core Metadata: dc:title, dc:subject, dc:creator, dc:contributor, dc:date, . . . ,
- RDF *reification* (cf. Slide 320): rdf:subject, rdf:predicate, rdf:object,
- user-defined annotation properties can be declared that describe non-OWL/DL relationships involving classes and properties.

9.4.2 [Aside] owl:hasKey (OWL 2)

Declaration of key attributes $(k_1, \ldots, k_n)$ amongst all objects of a class is a relevant issue in data modeling.

- a key allows for unambiguously identifying a resource amongst a certain subset of the domain,
- key conditions are *passive integrity constraints*
- in OWL, keys are not restricted to functional properties
  (i.e., SQL's UNIQUE is not required),
- values of key properties may be unknown for some instances; they might even be forbidden for some elements of the domain (e.g. using owl:maxCardinality 0 or owl:allValuesFrom owl:Nothing).
- note: InverseFunctionalProperty covers the simple case that $n = 1$ and the key is global.

owl:hasKey example

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:bla#>.
:name a owl:DatatypeProperty; a owl:FunctionalProperty.
:Country owl:hasKey (:carCode).
:DominicanRepublic a :Country; :carCode "D"; :name "Dominican Republic".
:Germany a :Country; :carCode "D"; :name "Germany".
```

[Filename: RDF/haskey.n3]

- the fragment is inconsistent.

OWL:hasKey for Non-Functional Properties

- keys are not restricted to functional properties:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:bla#>.
:District owl:hasKey (:code).
:Country owl:hasKey (:code).
:goettingen a :District; :name "Goettingen"; :code "GOE", "DUD", "HMÃU".
:leipzig a :District; :name "Leipzig"; :code "L".
:lahndillkreis a :District; :name "Lahn-Dill-Kreis"; :code "LDK", "DIL", "WZ", "L".
:luxembourg a :Country; :name "Luxembourg"; :code "L".
```

[Filename: RDF/key-mvd.n3]

```
prefix : <foo:bla#>
select ?D ?N ?C
from <file:key-mvd.n3>
where { ?X a ?D ; :name ?N; :code ?C }
```

[Filename: RDF/key-mvd.sparql]

- Lahn-Dill-Kreis and Leipzig are identified (LDK had “L” from 1977-1990).
- Luxembourg is not identified with them since the key definitions are local to districts vs. countries.

OWL:hasKey for Multi-Property-Keys

- consider triples about persons found in different Web sources.
- ABSOLUTELY BUGGY (27.7.2017) – it equates all four persons below:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:bla#>.
:Person owl:hasKey (:givenName :familyName).
 _:b1 a :Person; :givenName "John"; :familyName "Doe";  :age 35 .
 _:b2 a :Person; :givenName "John"; :familyName "Doe";  :address "Main Street 1" .
 _:b3 a :Person; :givenName "Mary"; :familyName "Doe";  :age 32; :address "Main Street 1" .
 _:b4 a :Person; :givenName "Donald"; :familyName "Trump";  :age 70; :address "White House" .
#:age a owl:FunctionalProperty.
```

[Filename: RDF/haskey2.n3]

```
prefix : <foo:bla#>
select ?X ?P ?Y
from <file:haskey2.n3>
where {?X a :Person ; ?P ?Y}
```

[Filename: RDF/haskey2.sparql]

[ASIDE/EXAMPLE] OWL:HASKEY AND NON-NAMED RESOURCES

Show that owl:hasKey ignores resources that are only implicitly known (OWL ontology see next slide):

- create an (infinite) sequence of implicitly known fathers ... all being persons and having the name “Adam”,
- guarantee that the sequence consists of different objects by making it irreflexive.
  (note: Transitivity and Irreflexivity are not allowed together, thus actually only every person is required to be different from his/her father – the grandfather might be the person again)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo:bla#>.
:Person owl:hasKey (:name) .
:name a owl:DatatypeProperty .
# :name a owl:InverseFunctionalProperty .  ## that would do it instead of hasKey
:father a owl:FunctionalProperty, owl:IrreflexiveProperty; rdfs:range :Person.
:bob a :Person; :father :john .
:john :name "John" .
:Adam owl:equivalentClass [ a owl:Restriction; owl:onProperty :name; owl:hasValue "Adam"] .
:Person rdfs:subClassOf
  [ a owl:Restriction; owl:onProperty :father; owl:someValuesFrom :Adam].
:JohnAdam owl:equivalentClass [ owl:intersectionOf ( :Adam
     [ a owl:Restriction; owl:onProperty :name; owl:hasValue "John"] ) ].
:hasFatherJohnAdam owl:equivalentClass [ a owl:Restriction;
     owl:onProperty :father; owl:someValuesFrom :JohnAdam ] .
:hasGrandpaAdam owl:equivalentClass [ a owl:Restriction; owl:onProperty :father;
    owl:someValuesFrom [ a owl:Restriction; owl:onProperty :father;
       owl:someValuesFrom :Adam ] ].
:AdamFatherAdam owl:equivalentClass [ owl:intersectionOf (:Adam
     [ a owl:Restriction; owl:onProperty :father;  owl:someValuesFrom :Adam] ) ] .
```

[Filename: RDF/forefathers-keys.n3]

[Aside/Example] owl:hasKey and Non-Named Resources

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo:bla#>
SELECT ?N ?A ?FA ?AFA ?GPA
FROM <forefathers-keys.n3>
WHERE {{ :bob :father [ :name ?N ] }
        # UNION { ?A :name "Adam" }   ## error/bug complains about anon(1)
        UNION { ?FA a :hasFatherJohnAdam }
        UNION { ?AFA a :AdamFatherAdam }
        UNION { ?GPA a :hasGrandpaAdam }}
```

[Filename: RDF/forefathers-keys.sparql]

- implicit nodes are not considered in the answers.
- owl:hasKey is not violated by the fact that several only implicitly known people are named “Adam”.
  Note that John, being Bob’s father, also gets the name “Adam”.

[Aside/Example] owl:hasKey and Non-Named Resources

Another example using multi-attribute keys (which could not be replaced by owl:InverseFunctionalProperty):

- nodes in a (x,y)-coordinate system; consider (10,10)
- insert a pointer to an implicit node (10,10).

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo:bla#>.
:XYThing owl:hasKey (:x :y).
:xy10 a :XYThing; :x 10; :y 10; :text "free".
:XYTen owl:intersectionOf ([ a owl:Restriction; owl:onProperty :x; owl:hasValue 10]
                           [ a owl:Restriction; owl:onProperty :y; owl:hasValue 10]
                [ a owl:Restriction; owl:onProperty :text; owl:hasValue "pointedTo"]).
:pointTo a owl:FunctionalProperty; rdfs:range :XYThing.
:foo a [ a owl:Restriction;
         owl:onProperty :pointTo; owl:onClass :XYTen; owl:qualifiedCardinality 1].
# :foo :pointTo :xyxy.   ## functionality of pointTo: makes :xyxy=(10,10) explicit
```

[Filename: RDF/easykeys-impl.n3]

[Aside/Example] owl:hasKey and Non-Named Resources (Cont'd)

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo:bla#>
SELECT ?CT ?Y ?T ?SameAsxyxy
FROM <easykeys-impl.n3>
WHERE {{ :foo :pointTo [ :text ?CT ] }
        UNION { ?Y :text ?T }
        UNION { [:text ?T] }
        UNION { :xyxy owl:sameAs ?SameAsxyxy }}
```

[Filename: RDF/easykeys-impl.sparql]

Implicit nodes are not considered in the answers.

- with last in line in source commented out: not much – the "pointTo" text is not answered, nothing is :sameAs.
- with last line commented in: the implicit node which is pointed to is equated with :xyxy, made explicit and then equated also with :xy10.

9.4.3 [Aside] OWL vs. RDF Lists

- RDF provides structures for representing lists by triples (cf. Slide 246): rdf:List, rdf:first, rdf:rest.
  These are *distinguished* classes/properties.
- OWL/reasoners have a still unclear relationship with these:
  - use of lists for its internal representation of owl:unionOf, owl:oneOf etc. (that are actually based on collections),
  - do or do not allow the user to query this internal representation,
  - ignore user-defined lists over usual resources.

[ASIDE] UNIONOF (ETC) AS TRIPLES: LISTS

- owl:unionOf (x y z), owl:oneOf (x y z) is actually only syntactic sugar for RDF lists.
- The following are equivalent:

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

:Male a owl:Class.
:Female a owl:Class.

:Person a owl:Class; owl:unionOf (:Male :Female).
:EqToPerson a owl:Class;
  owl:unionOf
  [ a rdf:List; rdf:first :Male;
    rdf:rest [ a rdf:List; rdf:first :Female; rdf:rest rdf:nil]].
:x a :Person.
```

[Filename: RDF/union-list.n3]

- jena -t -if union-list.n3: both in usual Turtle notation as owl:unionOf (:Male :Female).

[ASIDE] UNIONOF (ETC) AS TRIPLES (CONT'D)

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla#>
select ?C
from <file:union-list.n3>
where {:Person owl:equivalentClass ?C}
```

[Filename: RDF/union-list.sparql]

- jena -q -pellet -qf union-list.sparql: both are equivalent.

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla#>
select ?P1 ?P2 ?X ?Q ?R ?S ?T
from <file:union-list.n3>
where {{:Person owl:equivalentClass :EqToPerson} UNION
  {:Person ?P1 ?X . ?X ?Q ?R . OPTIONAL {?R ?S ?T}} UNION
  {:EqToPerson ?P2 ?X . ?X ?Q ?R} . OPTIONAL {?R ?S ?T}}
```

[Filename: RDF/union-list2.sparql]

- both have actually the same list structure
  (pellet2/nov 2008: fails; pellet 2.3/sept 2009: fails)

[ASIDE] REASONING OVER LISTS (PITFALLS!)

- rdf:first and rdf:rest are (partially) ignored for reasoning (at least by pellet?); they cannot be used for deriving other properties from it.
- they can even not be used in queries (since pellet2/nov 2008; before it just showed weird behavior)

```
prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla#>
select ?X ?Y ?Z
from <file:union-list.n3>
where {?X a rdf:List; rdf:first ?Y .
       OPTIONAL {?X rdf:rest ?Z}}
```

[Filename: RDF/union-list3.sparql]

- jena-tool with pellet2.3: OK.
- pellet2.3: NullPointerException.

[Aside] Extension of a class defined by a list

Given an RDF list as below, define an owl:Class :Invited which contains exactly the elements in the list (i.e., in the above sample data, :alice, :bob, :carol, :dave).

```
@prefix : <foo:bla#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
# Problem: when the real rdf namespace is used, rdf:first/rest are ignored
@prefix rdfL: <http://www.w3.org/1999/02/22-rdf-syntax-nsL#>.    # <<<<<<<<<<<<
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

:Invited a owl:Class.
:InvitationList rdfs:subClassOf rdfL:List.
:list1 a :InvitationList; rdfL:first :alice;
  rdfL:rest [a rdfL:List; rdfL:first :bob;
    rdfL:rest [a rdfL:List; rdfL:first :carol;
      rdfL:rest [a rdfL:List; rdfL:first :dave; rdfL:rest rdf:nil]]].

# rest of an InvitationList is also an InvitationList
:InvitationList owl:equivalentClass
  [a owl:Restriction;
    owl:onProperty rdfL:rest; owl:allValuesFrom :InvitationList],
  [ a owl:Restriction;
    owl:onProperty rdfL:first; owl:allValuesFrom :Invited].
```

[Filename: RDF/invitation-list.n3]

```
prefix : <foo:bla#>
select ?I
from <file:invitation-list.n3>
where {?I a :Invited}
```

[Filename: RDF/invitation-list.sparql]

9.5 Nominals: The O in SHOIQ

TBox vs. ABox

Description Logics Terminology

Clean separation between TBox and ABox vocabulary:

- TBox: RDFS/OWL vocabulary for information about classes and properties (further partitioned into definitions and axioms),
- ABox: Domain vocabulary and rdf:type.

RDF/RDF/OWL Ontologies

- Syntactically: allow to mix everything in a single set of triples.
- OWL-DL restriction: clean usage of individuals vs. classes
  - individuals only in application property triples (ABox)
  - classes only in context of RDFS/OWL built-ins (like (X a :Person) or (:hasChild rdfs:range :Person), etc.) (TBox)

Recall: Reification

- Reification treats a class (e.g. :Penguin) or a property as an individual (:Penguin a :Species)
- reification assigns properties from an application domain to classes and properties.
- useful when talking about metadata notions,
- risk: allows for paradoxes.

NOMINALS

- use individuals (that usually occur only in the ABox) in *specific positions* in the TBox:
- as individuals (that are often implemented in the reasoner as unary classes) with
  [a owl:Restriction; owl:onProperty *property*; owl:hasValue *object*]
  (the class of all things such that {?x *property object*} holds).
- in enumerated classes *class* owl:oneOf (o_1,...,o_n)
  (*class* is defined to be the set {o_1,...,o_n}).
  (Note: in owl:oneOf (o_1, ..., o_n), two items may be the same (open world))

USING NOMINALS: ITALIAN CITIES

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix it: <foo://bla#>.
it:italy owl:sameAs <http://www.semwebtech.org/mondial/10/countries/I/>.
it:ItalianCity a owl:Class; owl:intersectionOf
  (mon:City
   [a owl:Restriction; owl:onProperty mon:cityIn;
    owl:hasValue it:italy]).   # Nominal: an individual in a TBox axiom
```

[Filename: RDF/italiancities.n3]

```
prefix it: <foo://bla#>
select ?X
from <file:mondial-meta.n3>
from <file:mondial-europe.n3>
from <file:italiancities.n3>
where {?X a it:ItalianCity}
```

[Filename: RDF/italiancities.sparql]

- the query {?X :cityIn <http://www.semwebtech.org/mondial/10/countries/I/>} would be shorter, but here a class should be defined for further use ...

AN ONTOLOGY IN OWL

Consider the Italian-English-Ontology from Slide 52.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix f: <foo://bla#>.
f:Italian rdfs:subClassOf f:Person;
  owl:disjointWith f:English;
  owl:unionOf (f:Lazy f:LatinLover).
f:Lazy owl:disjointWith f:LatinLover.
f:English rdfs:subClassOf f:Person.
f:Gentleman rdfs:subClassOf f:English.
f:Hooligan rdfs:subClassOf f:English.
f:LatinLover rdfs:subClassOf f:Gentleman.
```

[Filename: RDF/italian-english.n3]

Class tree with jena -e:

```
owl:Thing
    bla:Person
        bla:English
            bla:Hooligan
            bla:Gentleman
        bla:Italian = bla:Lazy
owl:Nothing = bla:LatinLover
```

- LatinLover is empty, thus Italian $\equiv$ Lazy.

Italians and Englishmen (Cont'd)

- the conclusions apply to the instance level:

```
@prefix : <foo://bla#>.
:mario a :Italian.
```

[Filename: RDF/mario.n3]

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix : <foo://bla#>
select ?C
from <file:italian-english.n3>
from <file:mario.n3>
where {:mario rdf:type ?C}
```

[Filename: RDF/italian-english.sparql]

AN ONTOLOGY IN OWL

Consider the Italian-Professors-Ontology from Slide 53.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix it: <foo://bla#>.
@base <http://www.semwebtech.org/mondial/10/>.   # example for @base->Bolzano:
it:bolzano owl:sameAs <countries/I/provinces/Trentino-Alto+Adige/cities/Bolzano/>.
it:Italian owl:intersectionOf
  (it:Person
   [a owl:Restriction; owl:onProperty it:livesIn;
    owl:someValuesFrom it:ItalianCity]);
  owl:unionOf (it:Lazy it:Mafioso it:LatinLover).
it:Professor rdfs:subClassOf it:Person.
it:Lazy owl:disjointWith it:ItalianProf;
   owl:disjointWith it:Mafioso;
   owl:disjointWith it:LatinLover.
it:Mafioso owl:disjointWith it:ItalianProf;
   owl:disjointWith it:LatinLover.
it:ItalianProf owl:intersectionOf (it:Italian it:Professor).
it:enrico a it:Professor; it:livesIn it:bolzano.
```

[Filename: RDF/italian-prof.n3]

```
prefix : <foo://bla#>
select ?C
from <file:italian-prof.n3>
from <file:mondial-meta.n3>
from <file:mondial-europe.n3>
from <file:italiancities.n3>
where {:enrico a ?C}
```

[Filename: RDF/italian-prof.sparql]

ENUMERATED CLASSES: ONEOF

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
<bla:MontanunionMembers> owl:intersectionOf
 (mon:Country
  [owl:oneOf
   (<http://www.semwebtech.org/mondial/10/countries/NL/>
    <http://www.semwebtech.org/mondial/10/countries/B/>
    <http://www.semwebtech.org/mondial/10/countries/L/>
    <http://www.semwebtech.org/mondial/10/countries/F/>
    <http://www.semwebtech.org/mondial/10/countries/I/>
    <http://www.semwebtech.org/mondial/10/countries/D/>)]).
<bla:Result> owl:intersectionOf (mon:Organization
  [a owl:Restriction; owl:onProperty mon:hasMember;
   owl:someValuesFrom <bla:MontanunionMembers>]).
```

[Filename: RDF/montanunion.n3]

```
select ?X
from <file:montanunion.n3>
from <file:mondial-europe.n3>
from <file:mondial-meta.n3>
where {?X a <bla:Result>}
```

[RDF/montanunion.sparql]

- Query: all organizations that share a member with the Montanunion.
- Note: members of a owl:oneOf might be owl:sameAs (open world).

oneOf (Example Cont'd)

- previous example: "all organizations that share a member with the Montanunion." (DL: $x \in \exists$hasMember.MontanunionMembers)
- "all organizations where *all* members are also members of the Montanunion." (DL: $x \in \forall$hasMember.MontanunionMembers)
- The result is empty (although there is e.g. BeNeLux) due to open world: it is not known whether there may exist additional members of e.g. BeNeLux.
- Only if the membership of Benelux is "closed", results can be proven:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
<http://www.semwebtech.org/mondial/10/organizations/Benelux/>
 a [a owl:Restriction;
     owl:onProperty mon:hasMember; owl:cardinality 3].
<bla:SubsetOfMU> owl:intersectionOf (mon:Organization
  [a owl:Restriction; owl:onProperty mon:hasMember;
     owl:allValuesFrom <bla:MontanunionMembers>]).
mon:name a owl:FunctionalProperty. # not yet given in th
```

[Filename: RDF/montanunion2.n3]

```
select ?X
from <file:montanunion.n3>
from <file:montanunion2.n3>
from <file:mondial-europe.n3>
from <file:mondial-meta.n3>
where {?X a <bla:SubsetOfMU>}
```

[RDF/montanunion2.sparql]

oneOf (Example Cont'd)

- "all organizations that cover *all* members of the Montanunion."

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
<bla:EUMembers> owl:equivalentClass  [a owl:Restriction;
    owl:onProperty mon:isMember; owl:hasValue
    <http://www.semwebtech.org/mondial/10/organizations/EU/>].
```

[Filename: RDF/montanunion3.n3]

- note: such a class definition must be created (programmatically) for each organization.

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
select ?X # ?Y ?Z
from <file:montanunion.n3>
from <file:montanunion3.n3>
from <file:mondial-europe.n3>
from <file:mondial-meta.n3>
where {#{?Y a <bla:EUMembers>} UNION {?Z a <bla:MontanunionMembers>} UNION
       {<bla:MontanunionMembers> rdfs:subClassOf ?X}}
```

[Filename: RDF/montanunion3.sparql]

oneOf (Example Cont'd)

Previous example:

- only for one organization
- defined a class that contains all members of the organization
- not possible to define a *family of classes* – one class for each organization.
- this would require a *parameterized constructor*:

  "c_{org} is the set of all members of org"

  Second-Order Logic: each organization can be seen as a unary predicate (=set):

  $\forall Org : Org(c) \leftrightarrow$ hasMember(Org, c)

  or in F-Logic syntax: `C isa Org :- Org:organization[hasMember->C]`

  yields e.g.

  $I(eu) = \{germany, france, \ldots\}$,
  $I(nato) = \{usa, canada, germany, \ldots\}$

  Recall that "organization" itself is a predicate:

  $I(organization) = \{eu, nato, \ldots, \}$

  So this would require reification: organizations are both first-order-individuals and classes. Not allowed in OWL.

oneOf (Example Cont'd): Programmatically create classes and Annotation Properties

Using some (Java) framework, e.g., Jena (cf. Slide 545):

- the model (= the graph together with the reasoner) is accessible from Java
- iterate over all instances O of mon:Organization,
  - for each of them create the class definition for Omembers, add it to the model,
  - count its mon:hasMember edges and add the cardinality restriction to O,
  - with (cf. Slide 420)

    ```
    mon:hasMemberSet a owl:AnnotationProperty;
    ```

  - add O mon:hasMemberSet Omembers
- and state an appropriate SPARQL query. For answering not only the Omembers class URIs, but the names of the organization, mon:hasMemberSet has to be used. (Exercise in the "Semantic Web Lab Course")

CONVENIENCE CONSTRUCT: OWL:ALLDIFFERENT

- owl:oneOf defines a class as a closed set;
- in owl:oneOf (x_1, ..., x_n), two items may be the same (open world),

owl:AllDifferent

- Triples of the form :a owl:differentFrom :b state that two individuals are different.
  For a database with n elements, one needs
  $(n-1)+(n-2)+\ldots+2+1=\sum_{i=1..n} i = n\cdot(n+1)/2 = O(n^2)$ such statements.
- The –purely syntactical– convenience construct

  [a owl:AllDifferent; owl:members (r_1 r_2 ... r_n)]

  provides a shorthand notation.
  - it is *immediately* translated into the set of all statements
    {r_i owl:differentFrom r_j | $i \neq j \in 1..n$}
  - [a owl:AllDifferent; owl:members (...)]
    is to be understood as a (blank node) that acts as a *specification* that the listed things are different that does not actually exist in the model.

[SYNTAX] OWL:ALLDIFFERENT IN RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:f="foo://bla#"  xml:base="foo://bla#">
 <owl:Class rdf:about="Foo">
  <owl:equivalentClass> <owl:Class>
    <owl:oneOf rdf:parseType="Collection">
     <owl:Thing rdf:about="a"/>  <owl:Thing rdf:about="b"/>
     <owl:Thing rdf:about="c"/>  <owl:Thing rdf:about="d"/>
    </owl:oneOf>
   </owl:Class> </owl:equivalentClass>
 </owl:Class>
 <owl:AllDifferent>  <!-- use like a class, but is only a shorthand -->
    <owl:members rdf:parseType="Collection">
     <owl:Thing rdf:about="a"/>  <owl:Thing rdf:about="b"/>
     <owl:Thing rdf:about="c"/>  <owl:Thing rdf:about="d"/>
    </owl:members>
 </owl:AllDifferent>
 <owl:Thing rdf:about="a"> <owl:sameAs rdf:resource="b"/> </owl:Thing>
</rdf:RDF>
```

[Filename: RDF/alldiff.rdf]

```
prefix : <foo://bla#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?X ?P ?P2 ?V
from <file:alldiff.rdf>
where {?X a owl:AllDifferent ;
       ?P [?P2 ?V]}
```

[Filename: RDF/alldiffxml.sparql]

- AllDifferent is only intended as a kind of command to the application to add all pairwise “different-from” statements, it does not actually introduce itself as triples:
- querying {?X a owl:AllDifferent} is actually not intended.

[SYNTAX] OWL:ALLDIFFERENT IN TURTLE

Example:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:Foo owl:equivalentClass [ owl:oneOf (:a :b :c :d) ].
# both the following syntaxes are equivalent and correct:
[ a owl:AllDifferent; owl:members (:a :b)].
[] a owl:AllDifferent; owl:members (:c :d).
:a owl:sameAs :b.
# :b owl:sameAs :d.
```

[Filename: RDF/alldiff.n3]

```
prefix : <foo://bla#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?X ?Y
from <file:alldiff.n3>
where {?X a owl:AllDifferent ; ?P [?P2 ?V]}
```

[Filename: RDF/alldiff.sparql]

oneOf: A Test

- owl:oneOf defines a “closed set” (use with anonymous class; see below):
- note that in owl:oneOf ($x_1, \ldots, x_n$), two items may be the same (open world),
- optional owl:AllDifferent to guarantee that ($x_1, \ldots, x_n$) are pairwise distinct.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:Person owl:equivalentClass [ owl:oneOf (:john :alice :bob) ].
# :john owl:sameAs :alice.  # to show that it is consistent that they are the same
[] a owl:AllDifferent; owl:members (:john :alice :bob).  # to guarantee distinctness
# :name a owl:FunctionalProperty.  # this also guarantees distinctness ;)
:john :name "John".
:alice :name "Alice".
:bob :name "Bob".
:d a :Person.
:d owl:differentFrom :john, :alice.
# :d owl:differentFrom :bob.   ### adding this makes the ontology inconsistent
```

[Filename: RDF/three.n3]

- Who is :d?

oneOf: a Test (cont'd)

Who is :d?

- check the class tree:
  bla:Person - (bla:bob, bla:alice, bla:d, bla:john)
  The class tree does not indicate which of the "four" identifiers are the same.
- and ask it:

```
prefix : <foo://bla#>
select ?N
from <file:three.n3>
where {:d :name ?N}
```

[Filename: RDF/three.sparql]
The answer is ?N/"Bob".

A bug in Pellet

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo:bla/meta#>.

:Person a owl:Class ; owl:equivalentClass [ owl:oneOf (:john :alice) ] .
[ a owl:AllDifferent ; owl:members (:john :alice) ] .
:john a :Person .
:alice a :Person .
:xxx a :Person .
```

[Filename: RDF/one-of-bug.n3]

```
prefix :<foo:bla/meta#>
prefix owl: <http://www.w3.org/2002/07/owl#>
SELECT ?X ?Y ?Z
FROM <file:one-of-bug.n3>
WHERE {{?X owl:differentFrom ?Y} UNION {?X owl:sameAs ?Z}}
```

[Filename: RDF/one-of-bug.sparql]

xxx differentFrom alice (why?), but not the other way round, and also not john sameAs xxx.

X	Y	Z
xxx	alice	
john	alice	
alice	john	
xxx		xxx
john		john
alice		alice

9.6 Closing Parts of the Open World

- “forall items” is only applicable if additional items can be excluded ($\Rightarrow$ locally closed predicate/property),
- often, RDF data is generated from a database,
- certain predicates can be closed by defining restriction classes with maxCardinality.

Closing Parts of the Open World for owl:allValuesFrom

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
[ a :Male, :ThreeChildrenParent; :name "John";
   :hasChild [a :Female; :name "Alice"], [a :Male; :name "Bob"],
             [a :Female; :name "Carol"]].
[ a :Female, :TwoChildrenParent; :name "Sue";
   :hasChild [a :Female; :name "Anne";], [a :Female; :name "Barbara"]].
:name a owl:FunctionalProperty.
:OneChildParent owl:equivalentClass [a owl:Res
  owl:onProperty :hasChild; owl:cardinality 1].
:TwoChildrenParent owl:equivalentClass [a owl:
  owl:onProperty :hasChild; owl:cardinality 2].
:ThreeChildrenParent owl:equivalentClass [a owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 3].
:OnlyFemaleChildrenParent owl:equivalentClass [a owl:Restriction;
  owl:onProperty :hasChild; owl:allValuesFrom :Female].
```

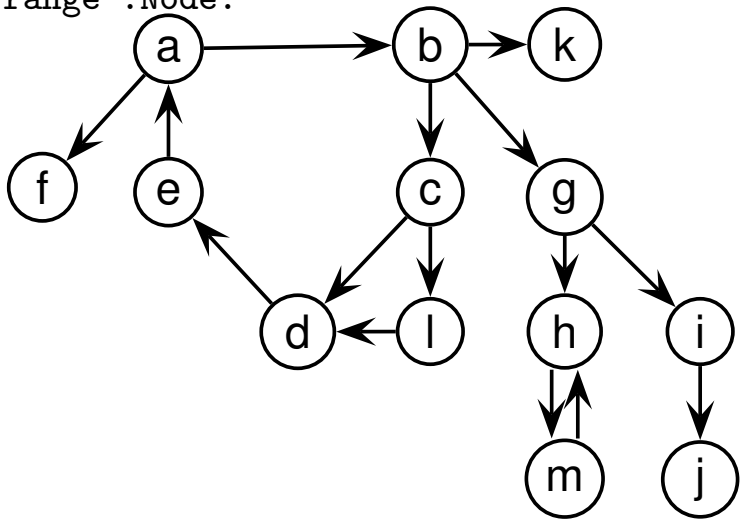
[Filename: RDF/allvaluesfrom.n3]

```
prefix : <foo://bla#>
select ?N
from <file:allvaluesfrom.n3>
where {?X :name ?N .
  ?X a :OnlyFemaleChildrenParent}
```

[Filename: RDF/allvaluesfrom.sparql]

EXAMPLE: WIN-MOVE-GAME IN OWL

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:Node a owl:Class; owl:equivalentClass
  [ a owl:Class; owl:oneOf (:a :b :c :d :e :f :g :h :i :j :k :l :m)].
[ a owl:AllDifferent; owl:members (:a :b :c :d :e :f :g :h :i :j :k :l :m)].
:edge a owl:ObjectProperty; rdfs:domain :Node; rdfs:range :Node.
:out a owl:DatatypeProperty, owl:FunctionalProperty.
:a a :Node; :out 2; :edge :b, :f.
:b a :Node; :out 3; :edge :c, :g, :k.
:c a :Node; :out 2; :edge :d, :l.
:d a :Node; :out 1; :edge :e.
:e a :Node; :out 1; :edge :a.
:f a :Node; :out 0 .
:g a :Node; :out 2; :edge :i, :h.
:h a :Node; :out 1; :edge :m.
:i a :Node; :out 1; :edge :j.
:j a :Node; :out 0 .
:k a :Node; :out 0 .
:l a :Node; :out 1; :edge :d.
:m a :Node; :out 1; :edge :h.
```



[Filename: RDF/winmove-graph.n3]

Win-Move-Game in OWL – the Game Axioms

“If a player cannot move, he loses.”

Which nodes are WinNodes, which one are LoseNodes (i.e., the player who has to move wins/loses)?

- if a player can move to some LoseNode (for the other), he will win.
- if a player can move only to WinNodes (for the other), he will lose.
- recall that there can be nodes that are neither WinNodes nor LoseNodes.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.

:WinNode a owl:Class; owl:intersectionOf ( :Node
  [a owl:Restriction; owl:onProperty :edge; owl:someValuesFrom :LoseNode]).
:LoseNode a owl:Class; owl:intersectionOf ( :Node
  [a owl:Restriction; owl:onProperty :edge; owl:allValuesFrom :WinNode]).
```

[Filename: RDF/winmove-axioms.n3]

Win-Move-Game in OWL – Closure

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:DeadEndNode a owl:Class;  rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 0],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 0].
:OneExitNode a owl:Class; rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 1],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 1].
:TwoExitsNode a owl:Class; rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 2],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 2].
:ThreeExitsNode a owl:Class; rdfs:subClassOf :Node;
  owl:equivalentClass [ a owl:Restriction; owl:onProperty :out; owl:hasValue 3],
                      [ a owl:Restriction; owl:onProperty :edge; owl:cardinality 3].
```

[Filename: RDF/winmove-closure.n3]

Win-Move-Game in OWL: DeadEndNodes

Prove that DeadEndNodes are LoseNodes:

- obvious: Player cannot move from there
- exercise: give a formal (Tableau) proof
- The OWL Reasoner does it:

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix : <foo://bla#>
select ?X
from <file:winmove-axioms.n3>
from <file:winmove-closure.n3>
where {:DeadEndNode rdfs:subClassOf :LoseNode}
```

[Filename: RDF/deadendnodes.sparql]

The answer contains an (empty) tuple which means “yes”.

Win-Move-Game instance solving in OWL

```
prefix : <foo://bla#>
select ?W ?L ?DE
from <file:winmove-graph.n3>
from <file:winmove-axioms.n3>
from <file:winmove-closure.n3>
where {{?W a :WinNode} UNION {?L a :LoseNode} UNION
       {?DE a :DeadEndNode}}
```

[Filename: RDF/winmove.sparql]

lose: f, k, j, e, l

win: c, a, i, b, d

The nodes g, h, and m are not contained in any of these sets $\rightarrow$ they are drawn positions.

Aside: Comparison with the Win-Move-Game in the “Deductive Databases” lecture: Solving vs. Reasoning

- With well-founded or stable semantics, concrete example cases of the win-move-game could be *solved*.
- With well-founded or stable semantics cannot make general *proofs* like DeadEndeNode $\sqsubseteq$ LoseNode.
- an OWL/DL *Reasoner* can do such *proofs*.

Exercise

- Is it possible to characterize DrawNodes in OWL?
  - 2 alternative variants:
    - using the game axioms/rules,
    - consider the possible values: win/lost/drawn,
  - test with *typical* minimal examples,
  - explain the results [DB Theory: compare also with well-founded and stable models].
- Is it possible to use SPARQL to find the drawn positions?

9.7 OWL 2 (W3C Recommendation since October 2009)

- OWL2 notions belong to the OWL namespace
  (aside: development proposal owl11 used a separate namespace)
- Syntactic Sugar: owl:disjointUnionOf, owl:AllDifferent, owl:AllDisjointClasses, owl:AllDisjointProperties, and negative assertions: ObjectPropertyAssertion vs. NegativeObjectPropertyAssertion
- User-defined datatypes (like XML Schema simple types).
- $\mathcal{SROIQ}$: Qualified cardinality restrictions (only for non-complex properties), local reflexivity restrictions (individuals that are related to themselves via the given property), reflexive, irreflexive, symmetric, and anti-symmetric properties (only for non-complex properties), disjoint properties (only for non-complex properties), Property chain inclusion axioms (e.g., SubPropertyOf(PropertyChain(owns hasPart) owns) asserts that if x owns y and y has a part z, then x owns z.
- $\mathcal{SROIQ}(D)$ is decidable.
  The Even More Irresistible $\mathcal{SROIQ}$. Ian Horrocks, Oliver Kutz, and Ulrike Sattler. In Principles of Knowledge Representation and Reasoning (KR 2006). AAAI Press, 2006. Available at `www.cs.man.ac.uk/~sattler/publications/sroiq-tr.pdf`.

QUALIFIED ROLE RESTRICTIONS

- extends owl:Restriction, owl:onProperty, owl:{min/max}QualifiedCardinality (int value) with owl:on{Class/DataRange} as result class/type.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .
:Dog a owl:Class.   :Cat a owl:Class.    :Cat owl:disjointWith :Dog.
:alice :name "Alice"; :hasAnimal :pluto, :struppi.
:john :name "John"; :hasAnimal :garfield, :odie.
:pluto a :Dog; :name "Pluto".
:struppi a :Dog; :name "Struppi".
:garfield a :Cat; :name "Garfield".
:odie a :Dog; :name "Odie".
:name a owl:FunctionalProperty.
:HasTwoAnimals owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:minCardinality 2].
:HasTwoCats owl:equivalentClass  [a owl:Restriction;
   owl:onProperty :hasAnimal; owl:onClass :Cat; owl:minQualifiedCardinality 2].
:HasTwoDogs owl:equivalentClass  [a owl:Restriction;
   owl:onProperty :hasAnimal; owl:onClass :Dog; owl:minQualifiedCardinality 2].
```

[Filename: RDF/cats-and-dogs.n3]

[Filename: RDF/cats-and-dogs.sparql]

```
prefix : <foo://bla#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
select ?X ?Y ?Z ?C   from <file:cats-and-dogs.n3>
where {{?X a :HasTwoCats} UNION
       {?Y a :HasTwoDogs} UNION
       {?Z a :HasTwoAnimals} UNION
       {?C rdfs:subClassOf :HasTwoAnimals}}
```

Qualified Role Restrictions – Another Test

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .
:alice :name "Alice"; :hasAnimal :pluto, :struppi.
:john :name "John"; :hasAnimal :garfield, :nermal, :odie.
:sue :hasAnimal :grizabella.                :grizabella :name "Grizabella".
:pluto a :Dog; :name "Pluto".               :struppi a :Dog; :name "Struppi".
:garfield a :Cat; :name "Garfield".         :nermal a :Cat; :name "Nermal".
:odie a :Dog; :name "Odie".
:name a owl:FunctionalProperty.
:Dog a owl:Class.    :Cat a owl:Class.      :Cat owl:disjointWith :Dog.
:HasAnimal owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:minCardinality 1].
:HasCat owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:onClass :Cat; owl:minQualifiedCardinality 1].
:HasDog owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasAnimal; owl:someValuesFrom :Dog].
```

[Filename: RDF/hasanimals.n3]

- export class tree:
  HasCat and HasDog are (non-disjoint) subclasses of HasAnimal.
- “owl:onClass X & owl:minQualifiedCardinality 1” is equivalent to “owl:someValuesFrom X”.
- “owl:minCardinality 1” alone is equivalent to “owl:someValuesFrom owl:Thing”.

Qualified Role Restrictions – Another Test

```
@prefix : <foo://bla#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
:TwoChildren owl:equivalentClass [a owl:Restriction;
  owl:onProperty :hasChild; owl:cardinality 2].
:ThreeMaleChildren owl:equivalentClass [a owl:Restriction;
  owl:onProperty :hasChild; owl:onClass :Male; owl:minQualifiedCardinality 3].
:TCTMC owl:equivalentClass
  [ owl:intersectionOf (:TwoChildren :ThreeMaleChildren) ].
```

[Filename: RDF/twochildren-threemale.n3]

- export class tree:
- note that the ontology is not inconsistent, but that simply TCTMC is derived to be equivalent to owl:Nothing.

OWL: Disjoint Union, AllDisjointClasses

... syntactic sugar for owl:unionOf and owl:disjointWith:

(only a simple test and syntax example for RDF/XML)

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:f="foo://bla#"
   xml:base="foo://bla#">
 <owl:Class rdf:about="Person">
  <owl:disjointUnionOf rdf:parseType="Collection">
   <owl:Class rdf:about="Male"/>
   <owl:Class rdf:about="Female"/>
  </owl:disjointUnionOf>
 </owl:Class>
 <f:Male rdf:about="John"/>
 <f:Female rdf:about="Mary"/>
 <!--<f:Female rdf:about="John"/>-->
</rdf:RDF>
```

[Filename: RDF/disjointunion.xml]

```
prefix f: <foo://bla#>
select ?X
from <file:disjointunion.xml>
where {?X a f:Person}
```

[Filename: RDF/disjointunion.sparql]

OWL: AllDisjointClasses

- General Case without union (similar to owl:AllDifferent):
  [a owl:AllDisjointClasses; owl:members (...)]
- Typical usages:
  - typically used if subclasses are disjoint specializations, but not every element of the superclass is an element of one of the specializations.
  - for the top classes of an ontology.

EXAMPLE: PARRICIDES IN GREEK MYTHODOLOGY (AND IN PSYCHOLOGY)

(from ESWC'07 SPARQL tutorial by Marcelo Arenas et al)

A parricide is a person who killed his/her father; <https://en.wikipedia.org/wiki/Oedipus>

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.          @prefix : <foo:greek#>.
:Person owl:disjointUnionOf (:Parricide :Non-Parricide).
  :iokaste a :Person; :hasChild :oedipus.
  :oedipus a :Person, :Parricide; :married-to :iokaste; :hasChild :polyneikes.
  :polyneikes a :Person; :hasChild :thersandros.
  :thersandros a :Person; a :Non-Parricide.
[ a owl:AllDifferent; owl:members (:iokaste :oedipus :polyneikes :thersandros)].
:Parent-of-Parricide owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :hasChild; owl:someValuesFrom :Parricide ].
:Parent-of-Non-Parricide owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :hasChild; owl:someValuesFrom :Non-Parricide ].
:Parent-of-Parricide-Grandparent-of-Non-Parricide owl:intersectionOf
  ([a owl:Restriction; owl:onProperty :hasChild; owl:someValuesFrom :Parricide]
   [a owl:Restriction;
     owl:onProperty :hasChild; owl:someValuesFrom :Parent-of-Non-Parricide]).
# :Parent-of-Parricide-Grandparent-of-Non-Parricide owl:equivalentClass owl:Nothing .
# makes it inconsistent                                  [Filename: RDF/parricide.n3]
```

Example (Cont'd)

```
prefix : <foo:greek#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?P ?NP ?PP ?PNP ?X
from <file:parricide.n3>
where {{?P a :Parricide} UNION
       {?NP a :Non-Parricide} UNION
       {?PP a :Parent-of-Parricide} UNION
       {?PNP a :Parent-of-Non-Parricide} UNION
       {?X a :Parent-of-Parricide-Grandparent-of-Non-Parricide}}
```

[Filename: RDF/parricide.sparql]

- No X reported.
- adding P.o.p.Gp.o.Np. $\equiv$ owl:Nothing makes the ontology inconsistent
- `:Parent-of-Parricide-Grandparent-of-Non-Parricide owl:equivalentClass owl:Nothing` can thus not be proven, but there is no direct query to the reasoner that it can be disproven.

$\Rightarrow$ needs an indirect way to prove a statement that is true if the class can be shown to be non-empty.

Example (Cont'd)

- ask Zeus whether Parent-of-Parricide-Grandparent-of-Non-Parricide is really non-empty:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo:greek#>.                                    [Filename: RDF/parricide2.n3]
:zeus :knows :iokaste, :oedipus, :polyneikes, :thersandros.
:KnowsPoPGonP owl:equivalentClass [ a owl:Restriction; owl:onProperty :knows;
  owl:someValuesFrom :Parent-of-Parricide-Grandparent-of-Non-Parricide ].
```

```
prefix : <foo:greek#>
select ?K ?X
from <file:parricide.n3>
from <file:parricide2.n3>                                  [Filename: RDF/parricide2.sparql]
where {{?K a :KnowsPoPGonP} UNION
       {?X a :Parent-of-Parricide-Grandparent-of-Non-Parricide}}
```

- Zeus is in K, i.e., he knows such a person (explicitly: he knows a person who must be a P.o.p.G.o.N.P),
- but neither SPARQL, nor Zeus know who that person is,
- it can be either Iokaste or Oedipus (depending on whether Polyneikes is a parricide, which nobody knows).

Example (Cont'd) – Exercise

Consider *absolutely strictly* the answer to `parricide2.sparql`.

- What has actually been logically proven by the answer?
- What *additional* human reasoning took place in the lecture when *interpreting* the answer to `parricide2.sparql` as "it can be either Iokaste or Oedipus (depending on whether Polyneikes is a parricide, which nobody knows)".
- complete the ontology and the SPARQL query in a way that the human resoning conclusions are mirrored in the setting.

Negative Assertions

- Assert that something is known *not* to hold:
  NegativeObjectPropertyAssertion and NegativeDataPropertyAssertion
- with owl:sourceIndividual, owl:assertionProperty, and owl:targetIndividual or owl:targetValue.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .
:john a :Person.
[ rdf:type owl:NegativePropertyAssertion;
  owl:sourceIndividual :john;
  owl:assertionProperty :lives;
  owl:targetIndividual :germany].
:German owl:equivalentClass [ a owl:Restriction;
  owl:onProperty :lives; owl:hasValue :germany ].
:NonGerman owl:complementOf :German.
```

[Filename: RDF/nongerman.n3]

```
prefix : <foo://bla#>
select ?P
from <file:nongerman.n3>
where {?P a :NonGerman}
```

[Filename: RDF/nongerman.sparql]

- John is derived to be a Non-German.

Comment on Negative Assertions

... are just syntactic sugar for a construct using complement classes (and actually implemented in the reasoner by this):

Any owl:NegativeObjectPropertyAssertion $\neg(x\ r\ y)$ is encoded as

- a restriction $R(r, y)$ based on owl:hasValue:
  $R(r, y) = \{x|(x\ r\ y)\}$
  (above: R(lives,germany) = :German)
- its complement $CompR(r, y) := \top \setminus R(r, y)$
  (above: CompR(lives,germany) = :NonGerman)
- and the assertion that $x \in CompR(r, y)$.
  (above: assert (:john a :NonGerman))

DATATYPES: HASVALUE WITH LITERAL VALUE

Characterize a class as the set of all things where a given property has a given value:

- all things in Mondial that have the name “Berlin”:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix : <foo:bla#>.
:Berlin owl:equivalentClass [ a owl:Restriction;
   owl:onProperty mon:name; owl:hasValue "Berlin"].
```

[Filename: RDF/has-literal-value.n3]

```
prefix : <foo:bla#>
select ?X
from <file:has-literal-value.n3>
from <file:mondial-europe.n3>
where {?X a :Berlin}
```

[Filename: RDF/has-literal-value.sparql]

- Often preferable: define an owl:DataRange (unary or enumeration), give it a url, and use some/allValuesFrom.

Enumerated Datatypes

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix uni: <foo://uni/>.
uni:graded a owl:FunctionalProperty;
 a owl:DatatypeProperty; rdfs:range uni:Grades.
uni:Grades a rdfs:Datatype;
 owl:equivalentClass [ a rdfs:Datatype;
  owl:oneOf ("1.0" "1.3" "1.7" "2.0" "2.3" "2.7" "3.0" "3.3" "3.7" "4.0") ] .
[ a uni:Thesis; uni:author <foo://bla/john>;
  uni:graded "2.5"].
```

[Filename: RDF/grades-one-of-namedset.n3]

- inconsistent: “2.5” does not belong to the allowed grades,
- note: “3” is also not allowed since “3” and “3.0” are different strings,
- see alternative next slide.

Enumerated Datatypes

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix uni: <foo://uni/>.
uni:graded a owl:FunctionalProperty;
 a owl:DatatypeProperty; rdfs:range [ a rdfs:Datatype;
  owl:oneOf (1 1.3 1.7 2.0 2.3 2.7 3 3.3 3.7 4) ] .
[ a uni:Thesis; uni:author <foo://bla/john>;
  uni:graded 2].
```

[Filename: RDF/grades-one-of-anonymous.n3]

```
prefix : <foo://uni/>
select ?X ?G
from <file:grades-one-of-anonymous.n3>
where {?X :graded ?G}
```

[Filename: RDF/grades-one-of-anonymous.sparql]

- grade 2.5 results in an inconsistency,
- internally (in case of an error message e.g.), the values are represented/handled as “2.3”^^xsd:decimal,
- parsing and output uses the default representation,
- both representations 2 and 2.0 are allowed.

OneOf on DataRange

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/meta#>.
:Male a owl:Class.    :Female a owl:Class.
:Person owl:disjointUnionOf (:Male :Female).
:MaleNames a rdfs:Datatype; owl:equivalentClass [ a rdfs:Datatype;
     owl:oneOf ("John"^^xsd:string "Bob"^^xsd:string) ] .
:FemaleNames a rdfs:Datatype; owl:equivalentClass [ a rdfs:Datatype;
    owl:oneOf ("Alice"^^xsd:string "Carol"^^xsd:string) ].
:Male a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :MaleNames])].
:Female a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :FemaleNames])].
:name a owl:FunctionalProperty; a owl:DatatypeProperty.
:john a :Person; :name "John"^^xsd:string.
:alice a :Person; :name "Alice"^^xsd:string.
```

[Filename: RDF/names.n3]

```
prefix : <foo://bla/meta#>
select ?C ?N
from <file:names.n3>
where {:john a ?C ; :name ?N}
```

[Filename: RDF/names.sparql]

Exercise

Consider again the ontology from the previous slide

- The name “Maria” is a female first name, but (mainly by catholics) also used as an additional first name for males, e.g. Rainer Maria Rilke (German poet, 1875-1926), José Maria Aznar (*1956, Spanish Prime Minister 1996-2004). Discuss the consequences on the ontology.
- Check what happens with names like “Kim” that can be both male and Female names.

REIFICATION

Reification: treat a class (or a property or a statement) as a thing:

- Male and Female are both classes and instances of class Gender

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix : <foo://bla/meta#>.
:Person owl:disjointUnionOf (:Male :Female).
:Male a :Gender.
:Female a :Gender.
:MaleNames owl:equivalentClass [ a rdfs:Datatype; owl:oneOf ("John" "Bob") ] .
:FemaleNames owl:equivalentClass [ a rdfs:Datatype; owl:oneOf ("Mary" "Alice") ].
:Male a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :MaleNames])].
:Female a owl:Class; owl:equivalentClass [owl:intersectionOf ( :Person
  [a owl:Restriction; owl:onProperty :name; owl:someValuesFrom :FemaleNames])].
:name a owl:FunctionalProperty; a owl:DatatypeProperty.
:john a :Person; :name "John".
:mary a :Person; :name "Mary".
```

[Filename: RDF/reification-class.n3]

```
prefix : <foo://bla/meta#>
select ?P ?N ?S
from <file:reification-class.n3>
where {{?S a :Gender .
        ?P a :Person ; a ?S ; :name ?N}}
```

[Filename: RDF/reification-class.sparql]

DATATYPES

- common built-ins from XML Schema: int, decimal, ..., date, time, datetime.
- “2”^^xsd:decimal is different from “2”^^xsd:int

```
prefix : <foo:bla#>
select ?X ?Y
from <file:decimal.n3>
where {?X :value ?Y}
```

[Filename: RDF/decimal.sparql]

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo:bla#>.
:value a owl:DatatypeProperty; rdfs:range xsd:decimal.
:foo :value "2"^^xsd:decimal; :value "1.0"^^xsd:decimal.
:foo :value "2.0"^^xsd:decimal; :value "2.3"^^xsd:decimal.
:foo :value "2"^^xsd:integer; :value "1"^^xsd:integer.
```

[Filename: RDF/decimal.n3]

- jena: returns 6 results: “2”^^xsd:decimal, 1.0, 2.0, 2.3, 1, 2
- pellet: returns 5 results: 1, 2, 2.3, 2.0, 1.0

DEFINING OWN DATATYPES

Two possibilities:

- use XML Schema xsd:simpleType definitions on the Web:
  - OWL reasoners parse+understand XML Schema simpleType declarations
  - adopt the DAML+OIL solution: datatype URI is constructed from the URI of the XML schema document and the local name of the simple type.
- OWL vocabulary to do the same as in XML Schema simpleTypes.

DATATYPES IN OWL

- use the XML Schema built-in types as resources (int and string must be supported; Pellet does also support decimal)
- rdfs:Datatype: cf. simple Types in XML schema; derived from the basic ones (e.g. xsd:int is an rdfs:Datatype)
- specified by
  - owl:onDatatype: from what datatype they are derived,
  - owl:withRestrictions is a list of restricting facets
  - facets as in XML Schema:
    xsd:{max/min}{In/Ex}clusive etc.
- similar to owl:Restrictions: define by
  myDatatypeName owl:equivalentClass [*datatypeSpec*].

DATA RANGES: ADULTS

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla#> .
:kate :name "Kate"; :age 62; :hasChild :john.
:john :name "John"; :age 35; :hasChild [:name "Alice"], [:name "Bob"; :age 8].
:hasChild rdfs:domain :Person; rdfs:range :Person.
:age a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:int.
:name a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:string.
:atLeast18T owl:equivalentClass
  [a rdfs:Datatype; owl:onDatatype xsd:int; owl:withRestrictions ( _:x1 )].
_:x1 xsd:minInclusive 18 .
:Adult owl:intersectionOf (:Person
  [ a owl:Restriction;
    owl:onProperty :age;
    owl:someValuesFrom :atLeast18T]).
:Child owl:intersectionOf (:Person
  [ owl:complementOf :Adult ]).
```

[Filename: RDF/adult.n3]

```
prefix : <foo://bla#>
select ?AN ?CN ?X ?Y
from <file:adult.n3>
where {{?A a :Adult; :name ?AN} UNION
       {?C a :Child; :name ?CN} UNION
       {?X :age ?Y}}
```

[Filename: RDF/adult.sparql]

An Example with Two QRRs

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla#> .
:kate :name "Kate"; :age 62; :hasChild :john, :sue.
:sue :name "Sue";  :age 32; :hasChild [:name "Barbara"].
:john :name "John";  :age 35;
      :hasChild :alice, [:name "Bob"; :age 8], [:name "Alice"; :age 10].
:frank :name "Frank"; :age 40; :hasChild [:age 18], [:age 13].
:hasChild rdfs:domain :Person; rdfs:range :Person.
:age a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:int.
:name a owl:FunctionalProperty; a owl:DatatypeProperty; rdfs:range xsd:string.
:atLeast18T owl:equivalentClass [a rdfs:Datatype;
  owl:onDatatype xsd:int; owl:withRestrictions ( [ xsd:minInclusive 18 ] ) ].
:Adult owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :age; owl:someValuesFrom :atLeast18T]).
:HasTwoAdultChildren  owl:equivalentClass [ a owl:Restriction;
    owl:onProperty :hasChild; owl:onClass :Adult; owl:minCardinality 2 ].
```

[Filename: RDF/adultchildren.n3]

```
prefix : <foo://bla#>
select ?AN ?N
from <file:adultchildren.n3>
where {{?A a :Adult; :name ?AN} UNION
       {?X a :HasTwoAdultChildren; :name ?N}}
```

[Filename: RDF/adultchildren.sparql]

DataRange Restriction for Geographical Coordinates

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.
:LongitudeT owl:equivalentClass [ a rdfs:Datatype; owl:onDatatype xsd:decimal;
   owl:withRestrictions ( [xsd:minExclusive -180] [xsd:maxInclusive 180] ) ] .
:LatitudeT owl:equivalentClass [ a rdfs:Datatype; owl:onDatatype xsd:decimal;
  owl:withRestrictions ( [ xsd:minInclusive -90] [xsd:maxInclusive 90] ) ] .
:EasternLongitudeT  owl:equivalentClass [a rdfs:Datatype;
  owl:onDatatype :LongitudeT; owl:withRestrictions ( [xsd:minInclusive 0] ) ] .
:EasternHemispherePlace owl:equivalentClass [a owl:Restriction;
  owl:onProperty mon:longitude; owl:someValuesFrom :EasternLongitudeT].
mon:longitude rdfs:range :LongitudeT.
mon:latitude rdfs:range :LatitudeT.
:Berlin a mon:City; :name "Berlin"; mon:longitude 13.3; mon:latitude 52.45 .
#:Atlantis a mon:City; :name "Atlantis"; mon:longitude -200; mon:latitude 100 .
:Lisbon a mon:City; :name "Lisbon"; mon:longitude -9.1; mon:latitude 38.7 .
```

[Filename: RDF/coordinates.n3]

```
prefix : <foo://bla/>
select ?N
from <file:coordinates.n3>
where {?X :name ?N .
  ?X a :EasternHemispherePlace}
```

[Filename: RDF/coordinates.sparql]

EXAMPLE: USING XSD DATATYPES

- [Does not work completely ...] Define simple dataypes in an XML Schema file:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="file:coordinates2.xsd">
 <xs:simpleType name="longitudeT">
  <xs:restriction base="xs:decimal">
   <xs:minExclusive value="-180"/>
   <xs:maxInclusive value="180"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="easternLongitude">
  <xs:restriction base="xs:decimal">
    <!-- note: base="longitudeT" would be nicer, but is not allowed when parsing from RDF -->
   <xs:minInclusive value="10"/>
   <xs:maxInclusive value="180"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="latitudeT">
  <xs:restriction base="xs:decimal">
   <xs:minInclusive value="-90"/>
   <xs:maxInclusive value="90"/>
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

[Filename: RDF/coordinates2.xsd]

... and now use the datatypes ...

```
<!DOCTYPE rdf:RDF [ <!ENTITY mon "http://www.semwebtech.org/mondial/10/meta#">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY Coords "file:coordinates2.xsd"> ]>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:mon="http://www.semwebtech.org/mondial/10/meta#">

 <!-- ****** IMPORTANT: ALL DATATYPES MUST BE MENTIONED TO BE PARSED ****** -->
 <rdfs:Datatype rdf:about="&Coords;#longitudeT"/>
 <rdfs:Datatype rdf:about="&Coords;#easternLongitude"/>
 <rdfs:Datatype rdf:about="&Coords;#latitudeT"/>
 <owl:Class rdf:about="&mon;EasternHemispherePlace">
 <owl:equivalentClass> <!-- again: don't give a uri to an owl:Restriction! -->
  <owl:Restriction>
   <owl:onProperty rdf:resource="&mon;longitude"/>
   <owl:someValuesFrom rdf:resource="&Coords;#easternLongitude"/>
  </owl:Restriction>
 </owl:equivalentClass>
 </owl:Class>

 <mon:City mon:name="Berlin">
  <mon:longitude rdf:datatype="&Coords;#longitudeT">13.3</mon:longitude>
  <mon:latitude rdf:datatype="&Coords;#latitudeT">52.45</mon:latitude> </mon:City>
 <mon:City mon:name="Lisbon">
  <mon:longitude rdf:datatype="&Coords;#longitudeT">-9.1</mon:longitude>
  <mon:latitude rdf:datatype="&Coords;#latitudeT">38.7</mon:latitude> </mon:City>
</rdf:RDF>
```

[Filename: RDF/coordinates2.rdf]

... and now to the query:

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?N
from <file:coordinates2.rdf>
where {?X :name ?N . ?X a :EasternHemispherePlace}
```

[Filename: RDF/coordinates2.sparql]

Comments

- the RDF file must “define” all used rdf:Datatypes to be parsed from the XML Schema file. (if <rdfs:Datatype rdf:about=“&Coords;#easternLongitude”/> is omitted, the result is empty)
- if a prohibited value, e.g. longitude=200 is given in the RDF file, it is rejected.
- the rdf:Datatype for mon:longitude and mon:latitude must be given, otherwise it is not recognized as a number (but it does not matter if xsd:int or coords:longitude is used).
- specifying rdfs:range for longitude and latitude *without* rdf:Datatype for mon:longitude and mon:latitude is even inconsistent!

QUALIFIED ROLE RESTRICTIONS: EXAMPLE

Example: Country with at least two cities with more than a million inhabitants.

- define “more than a million” as a rdfs:Datatype
- search for all BigCities (= more than 1000000 inhabitants)
- check -via Provinces- which countries have two such cities.

Example: Cont'd

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.

mon:population  rdfs:range xsd:int; a owl:FunctionalProperty. ## all cities are different.
_:Million a rdfs:Datatype; owl:onDatatype xsd:int; owl:withRestrictions (_:m1).
_:m1 xsd:minInclusive 1000000 .
:HasBigPopulation owl:equivalentClass [a owl:Restriction;
  owl:onProperty mon:population; owl:someValuesFrom _:Million].
:BigCity owl:intersectionOf (mon:City :HasBigPopulation).
:ProvinceWithBigCity owl:intersectionOf (mon:Province
  [a owl:Restriction; owl:onProperty mon:hasCity; owl:someValuesFrom :BigCity]).
:ProvinceWithTwoBigCities owl:intersectionOf (mon:Province ## europe: empty
  [a owl:Restriction; owl:onProperty mon:hasCity; owl:onClass :BigCity; owl:minCardinality 2]).
[owl:intersectionOf (mon:Country ## with 2 big cities, no provinces ## europe: empty
  [a owl:Restriction; owl:onProperty mon:hasCity; owl:onClass :BigCity; owl:minCardinality 2]);
  rdfs:subClassOf :CountryWithTwoBigCities].
[owl:intersectionOf (mon:Country ## with 2 provs with big cities ## TR,GB,E,R,UA,D,I,NL
  [a owl:Restriction; owl:onProperty mon:hasProvince; owl:onClass :ProvinceWithBigCity; owl:minCardinality 2]);
  rdfs:subClassOf :CountryWithTwoBigCities].
[owl:intersectionOf (mon:Country ## with a prov with 2 big cities ## europe: empty
  [a owl:Restriction; owl:onProperty mon:hasProvince; owl:someValuesFrom :ProvinceWithTwoBigCities]);
   rdfs:subClassOf :CountryWithTwoBigCities].
```

[Filename: RDF/bigcities.n3]

Example: Cont'd

```
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla/>.
:grmny a mon:Country; mon:hasCity :bln, :mch .
:bln a :BigCity; mon:population 3500000 .
:mch a :BigCity; mon:population 1500000 .
:frc a mon:Country; mon:hasProvince :ile, :prov .
:ile owl:differentFrom :prov.
:prs a mon:City; mon:cityIn :ile; mon:population 2000000 .
:mrs a mon:City; mon:cityIn :prov; mon:population 1500000
```

[Filename: RDF/dummy-cities.n3]

```
prefix : <foo://bla/>
select ?BC ?P1 ?P2 ?X
from <file:bigcities.n3>
from <file:dummy-cities.n3>
#from <file:mondial-europe.n3>
from <file:mondial-meta.n3>  ## population a owl:FunctionalProperty !
where {{?BC a :BigCity} UNION
       {?P1 a :ProvinceWithBigCity} UNION
       {?P2 a :ProvinceWithTwoBigCities} UNION
       {?X a :CountryWithTwoBigCities}}
```

note: when commenting out

`:ile owl:differentFrom :prov ,`

they *might* be the same object, but in both cases, France satisfies one of the conditions (one prov with two big cities, or two provinces with at least one big city each)

[Filename: RDF/bigcities.sparql]

9.8 OWL 2: More about Properties

- $\mathcal{SHIQ}$/OWL-DL concentrate on *concept* definitions ($\mathcal{SQ}$ portion),
  - The $\mathcal{H}$ allows for a hierarchy of *properties* as already provided by RDFS, the $\mathcal{I}$ allows for inverse.
- $\mathcal{SHOIQ}/\mathcal{SHOIQ}(D)$ add nominals and datatypes
  (i.e., provide database-oriented functionality for handling *instances*),
- $\mathcal{SROIQ}$ provides more expressiveness around *properties*.

Transitive and Symmetric Properties

- transitive: descendants (cf. Slide 241), train connections etc.
- symmetric: married

```
@prefix : <foo://bla#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
  [ :name "John";  :married [ :name "Mary" ]] .
  :married rdf:type owl:SymmetricProperty.
```

[Filename: RDF/symmetric-married.n3]

```
prefix : <foo://bla#>
select ?X ?Y
from <file:symmetric-married.n3>
where { [ :name ?X ; :married [ :name ?Y]] }
```

[Filename: RDF/symmetric-married.sparql]

Symmetric Properties

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .
:germany :borders :austria, :switzerland.
:borders a owl:SymmetricProperty.
```

[Filename: RDF/symmetricborders.n3]

```
prefix : <foo://bla#>
select ?X ?Y
from <file:symmetricborders.n3>
where {?X :borders ?Y}
```

[Filename: RDF/symmetricborders.sparql]

Reflexive Properties (OWL 2)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .
:john a :Person; :knows :mary; :hasChild :alice.
:knows a owl:ReflexiveProperty.
:germany a :Country.
```

[Filename: RDF/reflexive.n3]

```
prefix : <foo://bla#>
select ?X ?Y
from <file:reflexive.n3>
where {?X :knows ?Y}
```

[Filename: RDF/reflexive.sparql]

- only applied to individuals, but ... to all of them:
  John knows John, Alice knows Alice, and Germany knows Germany.

IRREFLEXIVE PROPERTIES

- irreflexive(rel): $\forall x : \neg rel(x, x)$.
- acts as constraint,
- but can also induce that two things must be different:
  $\forall x, y : rel(x, y) \rightarrow x \neq y$

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .
:john :hasAnimal :pluto, :garfield.
:pluto :bites :garfield.
# we exclude neurotic animals:
:bites a owl:IrreflexiveProperty.
:HasTwoAnimals  owl:equivalentClass
  [ a owl:Restriction;
    owl:onProperty :hasAnimal; owl:minCardinality 2 ].
```

[Filename: RDF/irreflexive.n3]

```
prefix : <foo://bla#>
select ?X ?Y ?Z
from <file:irreflexive.n3>
where {{?X :bites ?Y} UNION
       {?X :bites ?X} UNION
       {?Z a :HasTwoAnimals}}
```

[Filename: RDF/irreflexive.sparql]

- Pluto cannot be the same as Garfield.

ASYMMETRY

- asymmetric(rel): $\forall x, y : \neg rel(x, y) \vee \neg rel(y, x))$.
- acts as a constraint.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:rel a owl:AsymmetricProperty.
:a a :Node; :rel :b.
:b a :Node; :rel :c.
:c a :Node.
# :a owl:sameAs :b.
```

[Filename: RDF/asymmetry.n3]

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla#>
select ?X ?Y
from <file:asymmetry.n3>
where {?X a :Node; owl:differentFrom ?Y . ?Y a :Node}
```

[Filename: RDF/asymmetry.sparql]

- a,b,c, are not identified to be different, but any owl:sameAs makes the ontology inconsistent.

IRREFLEXIVE AND ASYMMETRIC PROPERTIES

- Motivated by the "Ascending, Descending" graphics by M.C.Escher
  `http://en.wikipedia.org/wiki/Ascending_and_Descending`

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla#>.
:Corner owl:oneOf (:a :b :c); rdfs:subClassOf
  [a owl:Restriction; owl:onProperty :higher; owl:cardinality 1].
[ a owl:AllDifferent; owl:members (:a :b :c)].
:higher rdfs:domain :Corner; rdfs:range :Corner.
:higher a owl:InverseFunctionalProperty. # necessary if there are more corners
:higher a owl:AsymmetricProperty; a owl:IrreflexiveProperty.
#:higher a owl:FunctionalProperty. ## redundant, note
:a :higher :b.
```

[Filename: RDF/escherstairs.n3]

- Solution: $a > b$, $b > c$, $c > a$ is the only model.

```
prefix : <foo://bla#>
select ?X ?Y
from <file:escherstairs.n3>
where {?X :higher ?Y}
```

[Filename: RDF/escherstairs.sparql]

Exercise

- what happens when the above program is the extended to four corners (:a :b :c :d)? Analyze the result also from the logical point of view.

DISJOINT PROPERTIES

- Syntax: ($prop_1$ owl:propertyDisjointWith $prop_2$)
- for more than 2 properties (similar to owl:AllDifferent):
  [a owl:AllDisjointProperties; owl:members (...)]

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla#>.

:name a owl:FunctionalProperty.
:hasCat rdfs:subPropertyOf :hasAnimal; rdfs:range :Cat.
:hasDog rdfs:subPropertyOf :hasAnimal; rdfs:range :Dog.
:hasCat owl:propertyDisjointWith :hasDog.

:alice :name "Alice"; :hasDog :pluto, :struppi.
:john :name "John"; :hasCat :garfield, :nermal; :hasDog :odie.
:sue :hasCat :grizabella.
#:sue :hasDog :grizabella.   ### test ########
:pluto a :Dog; :name "Pluto".
:struppi a :Dog; :name "Struppi".
:garfield a :Cat; :name "Garfield".
:nermal a :Cat; :name "Nermal".
:odie a :Dog; :name "Odie".
:grizabella :name "Grizabella".
```

[Filename: RDF/disjoint-properties.n3]

```
prefix : <foo://bla#>
select ?A ?B ?C ?D ?E ?F
from <file:disjoint-properties.n3>
where {{?X :name ?A; :hasCat/:name ?B} UNION
       {?X :name ?C; :hasDog/:name ?D} UNION
       {?X :name ?E; :hasAnimal/:name ?F}}
```

[Filename: RDF/disjoint-properties.sparql]

AT THE DECIDABILITY BORDER

Some combinations of advanced constructs in DL that are part of OWL 2 are not even decidable:

- $\mathcal{ALC}_{reg}$ with transitivity, composition and union is EXPTIME-complete
- the same when inverse roles and even cardinalities for *atomic* roles ($\mathcal{ALCQI}_{reg}$) are added (recall that inverse and transitive closure are important concepts in ontologies).
- The combination of *non-atomic* roles with cardinalities is in general undecidable.
- The same holds for Role-Value-Maps. Decidability is obtained only for Role-Value-Maps over *functional* roles.

CARDINALITIES ON ATOMIC ROLES

- a city can be the capital of at most one country (but also of one or more provinces)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/>.

:City a owl:Class; owl:equivalentClass
 [a owl:Restriction; owl:onProperty :isCapitalOf;
    owl:onClass :Country; owl:maxCardinality 1 ].

:name a owl:FunctionalProperty.
mon:C-Oslo a :City;
  :isCapitalOf mon:Norway, mon:P-Akershus, mon:P-Oslo.
mon:P-Akershus a :Province; :name "Akershus".
mon:P-Oslo a :Province; :name "Oslo".
mon:Norway a :Country; :name "Norway".
# mon:C-Oslo :isCapitalOf :foo.  :foo a :Country; :name "Foo".
```

[Filename: RDF/one-capital.n3]

- use jena -e to export class/instance tree

Across the Decidability Border

- Cardinality restrictions on complex (e.g. transitive) properties are not allowed (undecidable) $\Rightarrow$ rejected by the reasoner

Every city can be located in several provinces, but these must belong to the same country.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/>.

# Countries, Provinces, Cities:
:cityIn rdfs:subPropertyOf :belongsTo; rdfs:range :Province.
:isProvinceOf a owl:FunctionalProperty; rdfs:range :Country; rdfs:subPropertyOf :belongsTo.
:belongsTo a owl:TransitiveProperty; owl:inverseOf :hasProvOrCity.    # << trans.Prop <<<

:City a owl:Class; owl:equivalentClass
 [a owl:Restriction; owl:onProperty :belongsTo;                       # << cardinality <<
    owl:onClass :Country; owl:maxCardinality 1].

:name a owl:FunctionalProperty.
mon:C-Oslo a :City; :cityIn mon:P-Akershus, mon:P-Oslo.
mon:Norway a :Country; :name "Norway".
mon:P-Akershus a :Province; :isProvinceOf mon:Norway; :name "Akershus".
mon:P-Oslo a :Province; :isProvinceOf mon:Norway; :name "Oslo".
# mon:C-Oslo :isCapitalOf :foo.  :foo a :Country; :name "Foo".
```

[Filename: RDF/one-country.n3]

Detection of Potentially Undecidable Situations

Pellet does not accept combinations that can potentially be undecidable

The ontology is rejected by Pellet:

- Unsupported axiom: Ignoring transitivity axiom due to an existing cardinality restriction for property `http://www.semwebtech.org/mondial/10/meta#belongsTo`
- It is also rejected if

  ```
  :cityIn a owl:FunctionalProperty.
  :isProvinceOf a owl:FunctionalProperty.
  ```

  is added (which guarantees decidability).

FURTHER FEATURES OF OWL 2

- Role Chains/Property Chains: SubPropertyOf(PropertyChain(owns hasPart) owns)
  asserts that if x owns y and y has a part z, then x owns z.

  SubPropertyOf(PropertyChain(parent brother) uncle)
  asserts that the relationship "uncle" is a superset of "parent $\circ$ brother", i.e., the brothers of my parents are my uncles.

- Cross-property restrictions/role-value maps:
  (cf. draft at `http://www.w3.org/Submission/owl11-overview/`)
  - ObjectAllValuesFrom(likes knows =) describes the class of individuals who like all people they know (in DL syntax: the concept defined by the role value map (X.knows $\sqsubseteq$ X.likes)).
  - DataSomeValuesFrom(shoeSize IQ greaterThan)
    describes the class of individuals whose shoeSize is greater than their IQ (in DL syntax: the concept defined by the role value map (X.shoeSize$>$X.IQ)).

ROLE CHAINS

- (brotherOf $\circ$ hasChild) $\sqsubseteq$ uncleOf

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .

:name a owl:FunctionalProperty.
[ owl:propertyChain (:brotherOf :hasChild) ]
  rdfs:subPropertyOf :uncleOf.
:john a :Person; :brotherOf :sue.
:sue a :Person; :hasChild :anne, :barbara.
:anne :name "Anne".   :barbara :name "Barbara".
```

[Filename: RDF/uncle.n3]

```
prefix : <foo://bla#>
select ?U ?X
from <file:uncle.n3>
where {?U :uncleOf ?X}
```

[Filename: RDF/uncle.sparql]

Exercise

- Extend the above example: the husbands of sisters of parents of x are also x's uncles.

Syntax: Role Chains in RDF/XML

... as expected: a blank node that refers to an rdf:List which is an owl:subPropertyOf another property.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns="foo://bla#"
   xml:base="foo://bla#">
 <rdf:Description>
   <rdfs:subPropertyOf rdf:resource="#uncleOf"/>
   <owl:propertyChain>
     <rdf:List>
       <rdf:rest rdf:parseType="Collection">
         <owl:ObjectProperty rdf:about="#child"/>
       </rdf:rest>
       <rdf:first rdf:resource="#brotherOf"/>
     </rdf:List>
   </owl:propertyChain>
 </rdf:Description>
 <Person rdf:ID="sue">
  <hasChild rdf:resource="#anne"/>
  <hasChild rdf:resource="#barbara"/>
  <brother rdf:resource="#john"/>
 </Person>
 <Person rdf:ID="john">
  <brotherOf rdf:resource="#sue"/>
 </Person>
</rdf:RDF>
```

[Filename: RDF/uncle.rdf]

```
prefix : <foo://bla#>
select ?U ?X
from <file:uncle.rdf>
where {?U :uncleOf ?X}
```

[Filename: RDF/uncle2.sparql]

Role Chains

- propertyChains with 3 or more elements are allowed:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .

[ owl:propertyChain (:brotherOf :hasChild)]
  rdfs:subPropertyOf :uncleOf.
[ owl:propertyChain (:parent :brotherOf :hasChild)]
  rdfs:subPropertyOf :cousinOf.
# [ owl:propertyChain (:father)] rdfs:subPropertyOf :parent.  ## complains
# [ :uncleOf rdfs:subPropertyOf owl:propertyChain (:brotherOf :hasChild)]
#      is also not allowed (nullpointer error from inside pellet!)
:name a owl:FunctionalProperty.
:john a :Person; :brotherOf :sue.
:bob :parent :john.
:sue a :Person; :hasChild :anne, :barbara.
:anne :name "Anne".   :barbara :name "Barbara".
```

[Filename: RDF/propchain3-family.n3]

```
prefix : <foo://bla#>
select ?U ?X ?C
from <file:propchain3-family.n3>
where {{?U :uncleOf ?X}
       union {?C :cousinOf ?X}}
```

[Filename: RDF/propchain3-family.sparql]

Undecidable: Role Chains and Cardinalities

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#> .

:uncleOf a owl:ObjectProperty.   ### required !!!
[ ] rdfs:subPropertyOf :uncleOf;
    owl:propertyChain (:brotherOf :hasChild).

:name a owl:FunctionalProperty.
:john a :Person; :brotherOf :sue.
:sue a :Person; :hasChild :anne, :barbara.
:anne :name "Anne".   :barbara :name "Barbara".

:UncleOfMore a owl:Class; owl:equivalentClass
 [a owl:Restriction; owl:onProperty :uncleOf; owl:minCardinality 2].
```

[Filename: RDF/uncleOfMore.n3]

```
prefix : <foo://bla#>
select ?U ?X
from <file:uncleOfMore.n3>
where {{?U :uncleOf ?X} UNION
       {?U a :uncleOfMore}}
```

[Filename: RDF/uncleOfMore.sparql]

- pellet: Definition of uncle is ignored; result empty.
  WARNING - Unsupported axiom: Ignoring transitivity and/or complex subproperty axioms for uncleOf

SELF RESTRICIONS: $\{x \mid x\ r\ x\}$

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.
:Cyclic a owl:Class;
  owl:equivalentClass [ owl:intersectionOf
    (:Node [a owl:Restriction; owl:onProperty :to;
            owl:hasSelf "true"^^xsd:boolean ])].
:b a :Cyclic.
:a a :Node; :to :a, :b.
# :a a [ owl:complementOf :Cyclic ].
```

[Filename: RDF/cyclic.n3]

```
prefix : <foo://bla/>
select ?N ?N2
from <file:cyclic.n3>
where {{?N a :Cyclic} UNION
       {:a a :Cyclic} UNION
       {?N :to ?N2}}
```

[Filename: RDF/cyclic.sparql]

Self restrictions (Cont'd)

... just another example:

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla/>.
:NeuroticAnimal a owl:Class;
  owl:equivalentClass [ owl:intersectionOf
   ( :Animal
     [a owl:Restriction; owl:onProperty :bites; owl:hasSelf "true"^^xsd:boolean])].
:pluto a :Animal; :bites :pluto, :garfield.
:garfield a :NeuroticAnimal.
```

[Filename: RDF/neurotic.n3]

```
prefix : <foo://bla/>
select ?N ?N2
from <file:neurotic.n3>
where {{?N a :NeuroticAnimal} UNION
       {?N :bites ?N2}}
```

[Filename: RDF/neurotic.sparql]

Self restrictions (Cont'd)

... check for existence of cycles in a graph: Transitivity + SelfRestriction is not allowed:

WARNING: Unsupported axiom: Ignoring transitivity axiom due to an existing self restriction for property path

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix : <foo://bla#>.
:edge rdfs:subPropertyOf :path.    ### use a win-move game as input
:path a owl:TransitiveProperty.
:Cyclic a owl:Class;
  owl:equivalentClass [ owl:intersectionOf  ( :Node
     [a owl:Restriction; owl:onProperty :path; owl:hasSelf "true"^^xsd:boolean])].
```

[Filename: RDF/cyclic-transitive.n3]

```
prefix : <foo://bla#>
select ?X ?Y ?N
from <file:cyclic-transitive.n3>
from <file:winmove-graph.n3>
where {{?X :path ?Y} UNION {?N a :Cyclic}}
```

[Filename: RDF/cyclic-transitive.sparql]

9.9 DL and OWL Proving and Query Answering

- Tableau provers use refutation techniques:
  Given an ontology formalization Φ,
  prove $\Phi \models \varphi$ by starting a tableau over $\Phi \wedge \neg\varphi$ and trying to close it.

For that, it is well-suited for *testing* if something holds:

- consistency of a concept definition:
  $KB \models C \equiv \bot \;\Leftrightarrow\; KB \cup \{C(a)\}$ for a new constant a is unsatisfiable.
  (note: often C is a complex DL class definition that encodes the answers to a query)
- concept containment:
  $KB \models C \sqsubseteq D \;\Leftrightarrow\; KB \models (C \sqcap \neg D) \equiv \bot$.
- concept equivalence:
  $KB \models C \equiv D \;\Leftrightarrow\; KB \models C \sqsubseteq D$ and $KB \models D \sqsubseteq C$.
- concept membership (for a given individual a):
  $KB \models C(a) \;\Leftrightarrow\; KB \cup \{\neg C(a)\}$ is unsatisfiable.

TABLEAU EXPANSION RULES FOR DL

- DL: uses a tableau *without free variables*. Expansion of universally quantified formulas takes only place for *constants* that are actually introduced.
- makes it more similar to Model Checking
- actually, not the tableau is generated completely, but branches are investigated by backtracking.

$(C \sqcap D)(s)$	Add $C(s)$ and $D(s)$ to the branch.
$(C \sqcup D)(s)$	Add two branches, one with $C(s)$, the other with $D(s)$.
$\exists R.C(s)$	Add $R(s,c)$ and $C(c)$ where c is a new constant symbol.
$\forall R.C(s)$	Add $C(t)$ whenever $R(s,t)$ is on the tableau (requires bookkeeping).
$\geq nR.C(s)$	Add $R(s,c_1),\ldots,R(s,c_n),C(c_1),\ldots,C(c_n)$ and $c_i \neq c_j$ where c_i are new.
$\leq nR.C(s)$	Bookkeeping about $\{c \mid R(s,c)\}$ and $\{c \mid C(c)\}$. Whenever more than n, then add branches with all combinations $c_i = c_j$. Continue bookkeeping.
$C \sqsubseteq D$	For each s recursively add two branches with $\neg C(s)$ and $D(s)$.
Closure	Close a branch whenever $A(s)$ and $\neg A(s)$ occur.

QUERY ANSWERING IN DL AND OWL

Query answering requires to find all answer bindings to variables.

- find all X such that $KB \models C(X)$.
- find all D such that $KB \models D \sqsubseteq C$.

Start a tableau and collect substitutions that close branches:

- start with $KB \cup \{\neg C(X)\}$.
- collect substitutions for X for which the tableau closes.
- without free variables: generate a new $\neg C(s)$ whenever any rule introduces a constant s.
  (= check if that s is an answer)
- harder to implement.
  Not always all answers are found by the current implementations.
- help the system by not only asking "{?X :age ?Y}", but pruning the search space by "{?X a :Person; :age ?Y}".

DL TABLEAUX: EXAMPLES

Who are John's children?

hasChild(kate,john)
name(john,"John")
hasChild(john,alice)
name(alice,"Alice")
hasChild(john,bob)
name(bob,"Bob")

Query: ?- hasChild(john,X).

¬ hasChild(john,X)

$\square\{X_1 \leftarrow$ alice$\}$

$\square\{X_2 \leftarrow$ bob$\}$

What are the names of John's children?

hasChild(john,alice)
hasChild(john,bob)
name(john,"John")
name(alice,"Alice")
name(bob,"Bob")

Query: ?- hasChild(john,_X), name(_X,N).

$\neg$(hasChild(john,X) $\wedge$ name(X,N))

$\neg$(hasChild(john,X))

Try $\square\{X_1 \leftarrow$ alice$\}$

Try $\square\{X_2 \leftarrow$ bob$\}$

$\neg$name(X,N)

for X_1 and X_2:

X_1

¬ name(alice,N)

$N_1 \leftarrow$"Alice"

X_2

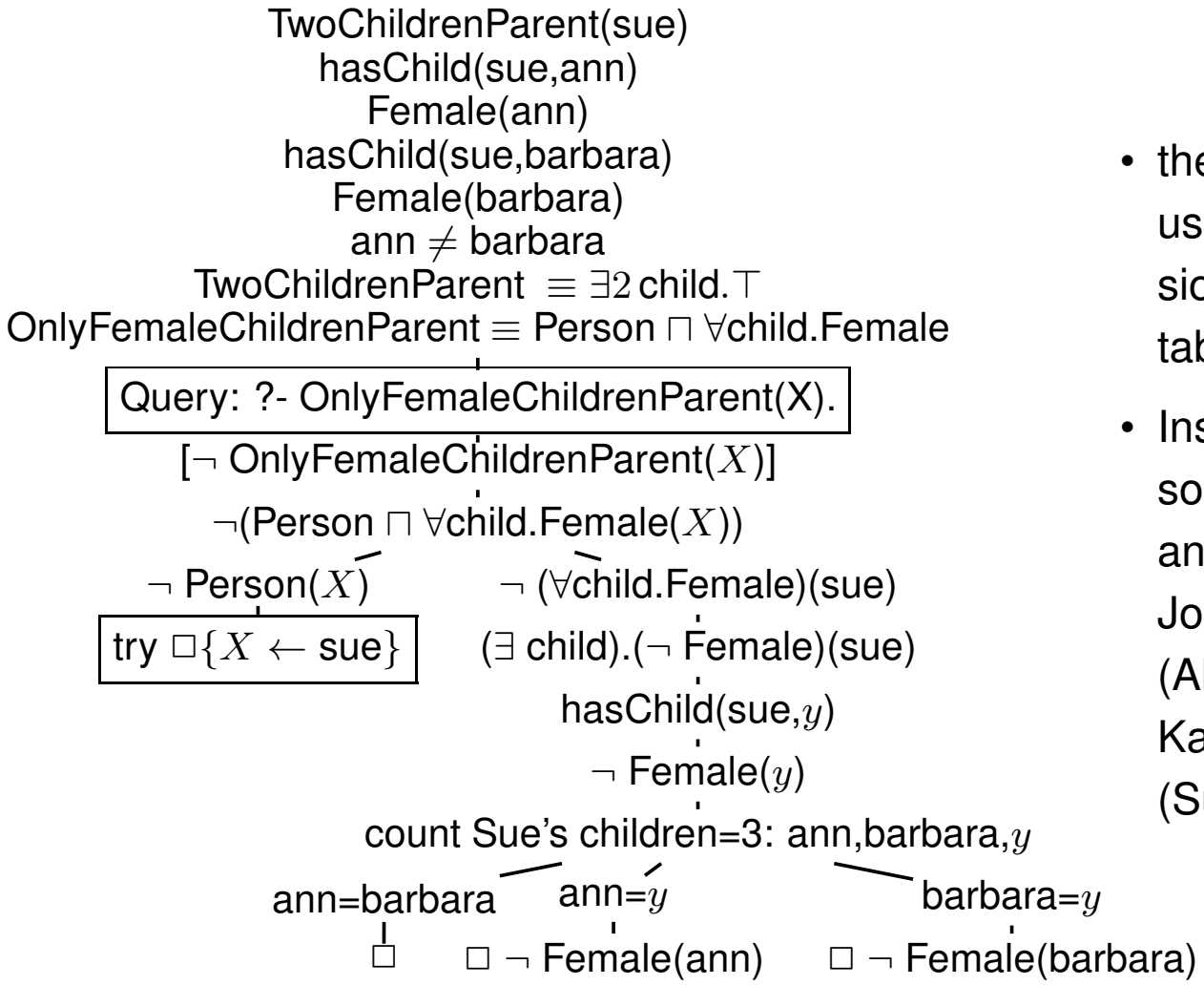
¬ name(bob,N)

$N_2 \leftarrow$"Bob"

- Note: one could try close the right branch with $X_0 \leftarrow$john and $N_0 \leftarrow$"John", but for that, the left branch will not close.
- Internal Strategy: don't explicitly close with X_i.
  Instead prepare complete tableau and compute closing *relational algebra expression*:
  (σ[$1=john](hasChild($1,X))) $\times$ name(X,N)

DL TABLEAUX: EXAMPLES

Consider the "Only female children" example from Slide 453.



- the negated query can be used for leading the expansion, but not for closing the tableau.
- Instead of X, all other persons are also tried to derive answers:
  John: tableau does not close (Alice)
  Kate: tableau does not close (Sue)

DL TABLEAUX: A MORE INVOLVED EXAMPLE

Consider again the Escher Stairs example (Slide 495).

(1) Corner = AllDifferent(a,b,c)
(2) cardinality: Corner $\sqsubseteq \exists 1$ higher.$\top$
(3) domain: Corner $\sqsupseteq \exists$ higher.$\top$
(4) range: $\top \sqsubseteq \forall$ higher.Corner
(5) AntiSymmetric(higher)
(6) Irreflexive(higher)
(7) higher(a,b)

Query: ?- higher(X,Y).

[$\neg$ higher(X,Y)]

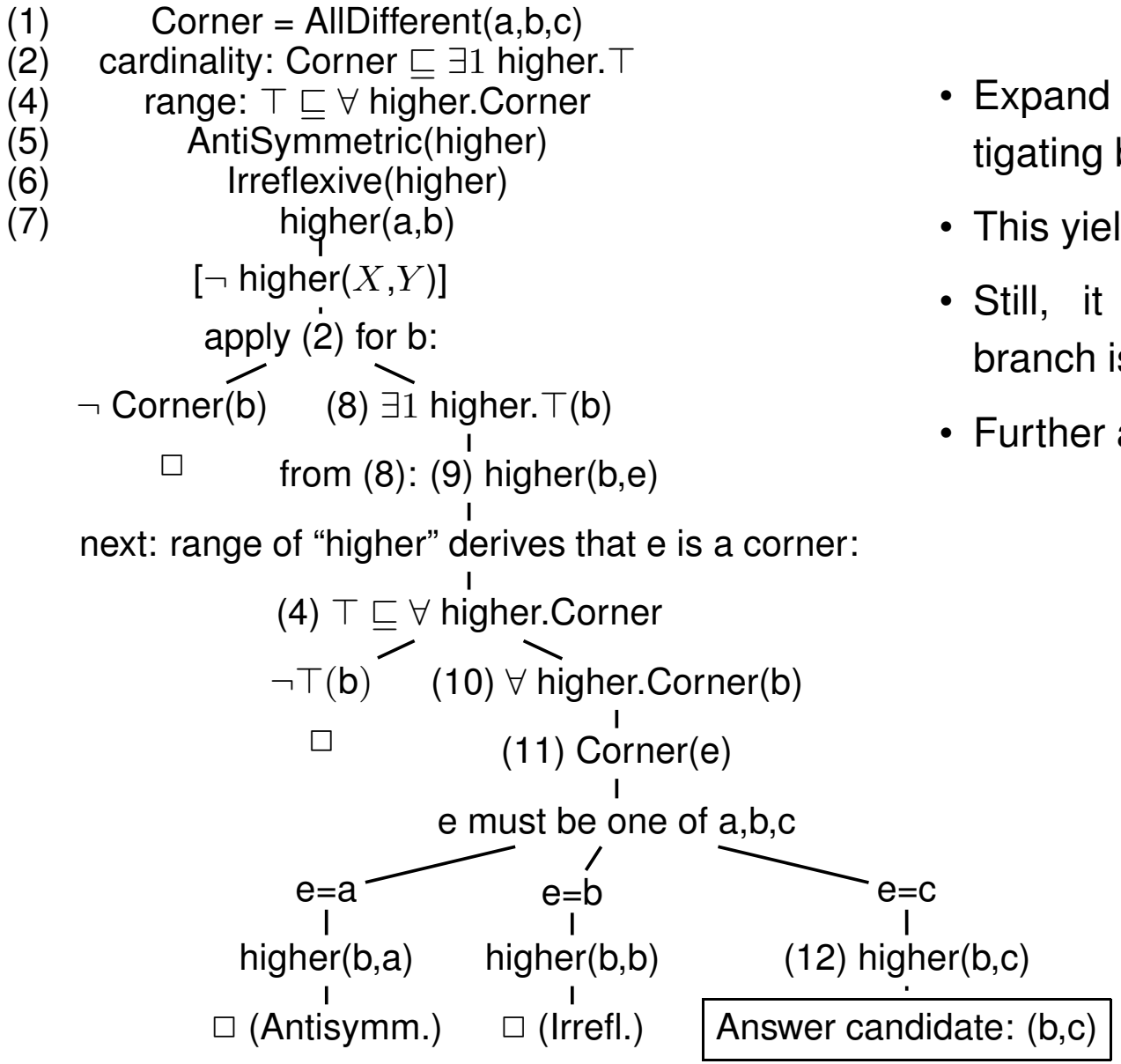
First Answer Candidate:
with (7) X$\leftarrow$a, Y$\leftarrow$ b
Try further answers ...

- The negated query can be used for leading the expansion, but not for closing the tableau. The first answer candidate is higher(a,b) – which was given in the input.
- Show that the developing model is consistent,
- and try to find additional answer candidates.

(2) can be applied for any constant occurring in the branch.

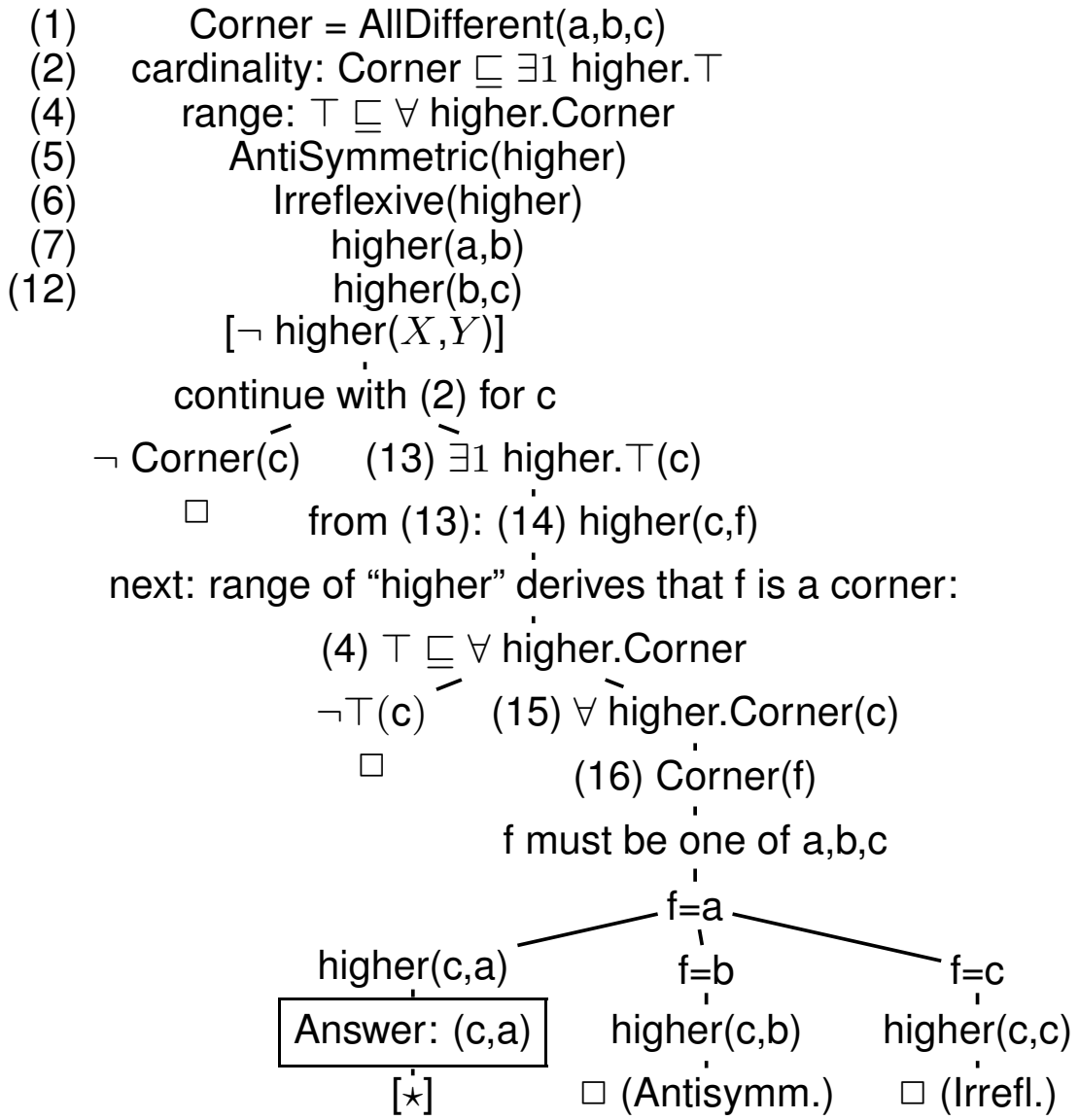
- Choose “b” since it is already used in another fact and search for further answers in this model.

Escher stairs tableau: continue with (2) for b



- Expand the branch (=model) by investigating b.
- This yields another answer candidate.
- Still, it must be checked that the branch is not inconsistent.
- Further answers will be found.

Escher stairs tableau: continue with (2) for c



- The branch [⋆] cannot be closed.
- The set of formulas on this branch is consistent and describes a model.
- The answers to ?- higher(X,Y) in this model are (a,b), (b,c), and (c,a).

REQUIREMENTS ON (NOT ONLY DL) TABLEAU STRATEGIES

- select most promising formula to be expanded next
  - based on coincident constants,
  - "selectivity" of conditions,
  - α-rules non-branching before β-rules (branching).
- non-closing branches: know when to stop and return answer matches
  - "saturated" branches: expansion does not add new formulas,
  - do not expand irrelevant formulas at all.

DL TABLEAUX: SO FAR, SO GOOD ...

Consider the axiom

$$\text{Person} \sqsubseteq \exists \text{hasParent.Person}$$

The tableau generation does not terminate.

Blocking

- a constant s_2 is introduced as an existential filler from expanding a fact about constant s_1,
- the knowledge about s_1 and s_2 is *saturated* (i.e., nothing new about them can be derived),
- and the same facts are known about s_1 and s_2 except the above existential chain,
- then *block* s_2 from application of the existential formula (which would just create another same thing).
- Such blocking can be done for every existentially introduced thing, and it has only to be dropped if differences between it and its "predecessor" are derived.
- Such ontologies can be used. Queries only return instances in the "relevant" finite portion.

BLOCKING

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla#>.
:kate a :Person; :name "Kate"; :hasChild :john.
:john a :Person; :name "John"; :hasChild :alice.
:alice a :Person; :name "Alice".
:hasChild rdfs:domain :Parent;
       owl:inverseOf :hasParent.
:Person rdfs:subClassOf
 [a owl:Restriction;  owl:onProperty :hasParent;  owl:cardinality 2].
:Parent owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].
:Grandparent owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasChild; owl:someValuesFrom :Parent].
:HasParent owl:equivalentClass
 [a owl:Restriction; owl:onProperty :hasParent; owl:someValuesFrom owl:Thing].
```

[Filename: RDF/infinite-parents.n3]

Blocking (cont'd)

```
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix : <foo://bla#>
select ?A ?B ?C ?R ?X
from <file:infinite-parents.n3>
where {{?A a :Parent} UNION
       {?B a :Grandparent} UNION
       {?C a :HasParent} UNION  ## kate has a parent ...
       {:hasParent rdfs:range ?R} UNION
       {:kate :parent ?X}} # ... which is not output
```

[Filename: RDF/infinite-parents.sparql]

- The tableau strategy of pellet correctly blocks the generation of (useless) blank nodes.
- Note: when trying to count "how many persons must exist", or "can we prove that at least 10 persons exist" would require to exclude that there is a cycle in the parents' chain.

EXERCISE

Write RDF/OWL instances:

- John has two children in school, they are in the 3nd and 5th year. Children in the first year are 6 years old, those in the 2nd year are 7 years old, and so on. There are 12 years of school.
- Alice is a daugther of John. She is 10 years old.
- an “ideal family” consists of a father, a mother, and they have 2 children, a son and a daughter, and a dog.
- John’s family is an “ideal family”.
- Bob is John’s son.

Feed them into the Jena tool, activate the reasoner.

- How old is Bob?
- which of the above information can be omitted without losing information how old Bob is?

9.10 Open World and Closed World: OWL/DL/Tableaux/Logic and SPARQL

- OWL/DL reasoning: OWA.
  Everything that can neither be proven nor disproven is unknown
- SPARQL queries/algebraic evaluation: CWA
  BGPs that do not match (not proven to be true, i.e. false or unknown) are considered as "no answer"

SPARQL CWA AND OWL OWA: POSSIBLE – IF NOT IMPOSSIBLE

$\Rightarrow$ Use SPARQL to check what cannot be *proven* by using FILTER NOT EXISTS { *query* }:
If the negation of some formula φ cannot be proven – then φ is at least possible, i.e, there exists a model that makes φ true.

- Limited expressiveness $\neg\varphi$ must be OWL-DL-expressible.
  (means: wrt. stable models [Deductive Databases Lecture], where any possible solution can be described)
- Consider again Slide 398 for an earlier example.

SPARQL NOT EXISTS { $\neg\varphi$ } for checking possibility of φ

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://bla#>.
:Childless owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :hasChild; owl:maxCardinality 0]).
:Parent owl:intersectionOf (:Person
  [ a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1]).
:john a :Person; :hasChild :alice, :bob.
:alice a :Person.  :bob a :Person.
```

[Filename: RDF/childless-small.n3]

```
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix : <foo://bla#>
select ?X ?C
from <file:childless-small.n3>
where { ?X a :Person . ?C a owl:Class; rdfs:subClassOf :Person
        FILTER NOT EXISTS {?X a ?C}}
```

[Filename: RDF/childless-small.sparql]

- John: only possible that he is a parent;
  for alice and bob, it is possible to be a parent or to be childless.

SPARQL CWA and OWL OWA: Possible – if not impossible: A scenario

- three rooms: bedroom, livingroom, guestroom
- some furniture: beds, a wardrobe, tables, chairs
- specification how many of these furniture can be placed in the rooms
- task: find out what can be placed where

Scenario

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix : <foo://rooms#>.
:in a owl:ObjectProperty, owl:FunctionalProperty; owl:inverseOf :has;
  rdfs:domain :Furniture; rdfs:range :Room.
:Room owl:oneOf (:bedroom :livingroom :guestroom).
[] a owl:AllDifferent; owl:members (:bedroom :livingroom :guestroom).
:bedroom a :Room,
  [a owl:Restriction; owl:onProperty :has; owl:onClass :Bed; owl:qualifiedCardinality 1],
  [a owl:Restriction; owl:onProperty :has; owl:onClass :Wardrobe; owl:qualifiedCardinality 1],
  [a owl:Restriction; owl:onProperty :has; owl:onClass :Chair; owl:qualifiedCardinality 1],
  [a owl:Restriction; owl:onProperty :has; owl:onClass :Table; owl:maxQualifiedCardinality 0].
# :bedroom  :has :bed1 . # comment in or out ...
:guestroom a :Room,
  [a owl:Restriction; owl:onProperty :has; owl:onClass :Table; owl:maxQualifiedCardinality 0].
:livingroom a :Room,
  [a owl:Restriction; owl:onProperty :has; owl:onClass :Bed; owl:maxQualifiedCardinality 0],
  [a owl:Restriction; owl:onProperty :has; owl:onClass :Chair; owl:qualifiedCardinality 4].

:Furniture a owl:Class;
  owl:disjointUnionOf (:Bed :Wardrobe :Table :Chair);
  owl:equivalentClass [a owl:Restriction; owl:onProperty :in; owl:cardinality 1].
```

```
:Bed owl:oneOf (:bed1 :bed2 :bed3).  ### one in bedroom, none in livingr. -> two in guestroom
[] a owl:AllDifferent; owl:members (:bed1 :bed2 :bed3).
:Wardrobe owl:oneOf (:wr1).
:Table owl:oneOf (:t1).   ### only one. must be in livingroom -> no in bedroom.
:Chair owl:oneOf (:c1 :c2 :c3 :c4 :c5).
[] a owl:AllDifferent; owl:members (:c1 :c2 :c3 :c4 :c5).
  ### one must be in bedroom, 4 in livingroom, no one remains for guestroom

:InBedroom a owl:Class; owl:equivalentClass [ a owl:Restriction;
   owl:onProperty :in; owl:hasValue :bedroom ].
:InGuestroom a owl:Class; owl:equivalentClass [ a owl:Restriction;
   owl:onProperty :in; owl:hasValue :guestroom ].
:InLivingroom a owl:Class; owl:equivalentClass [ a owl:Restriction;
   owl:onProperty :in; owl:hasValue :livingroom ].
:NotInBedroom a owl:Class; owl:equivalentClass
   [ owl:intersectionOf (:Furniture [ owl:complementOf :InBedroom])].
:NotInLivingroom a owl:Class; owl:equivalentClass
   [ owl:intersectionOf (:Furniture [ owl:complementOf :InLivingroom])].
:NotInGuestroom a owl:Class; owl:equivalentClass
   [ owl:intersectionOf (:Furniture [ owl:complementOf :InGuestroom])].

## for the queries:
:RoomWithChair owl:equivalentClass
  [a owl:Restriction; owl:onProperty :has; owl:someValuesFrom :Chair].
## :guestroom a :RoomWithChair.   ## makes it inconsistent
:RoomWithoutChair owl:equivalentClass [a owl:Restriction;
  owl:onProperty :has; owl:onClass :Chair; owl:maxQualifiedCardinality 0].
:RoomWithTwoBeds owl:equivalentClass [a owl:Restriction;
  owl:onProperty :has; owl:onClass :Bed; owl:qualifiedCardinality 2].
```

[Filename: RDF/rooms.n3]

Scenario (Cont'd)

```
prefix : <foo://rooms#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?X ?Room ?InR ?Y ?InterpretAsMaybeInR
from <file:rooms.n3>
where {{ ?X :in ?Room}  UNION
  { ?X a :Furniture, ?InR . ?InR rdfs:subClassOf :Furniture  .
       FILTER contains(str(?InR),"In")}
    UNION
  { ?Y a :Furniture . ?NotInR rdfs:subClassOf :Furniture .
       FILTER contains(str(?NotInR),"NotIn") .
    FILTER NOT EXISTS { ?Y a ?NotInR .}
    bind (?NotInR as ?InterpretAsMaybeInR)
  }}
order by ?R ?InR ?NotInR
```

[Filename: RDF/rooms.sparql]

- The table must be in the livingroom,
- among bed1, bed2, bed3, one is in the bedroom and two are in the guestroom.

Scenario (Cont'd)

- are there two beds in the guestroom? (yes)
- is it possible that there is some chair in the guestroom?
  (no - it can be derived that the guestroom is a room without chair)
- is it possible that chair1 is in the bedroom? (yes)

```
prefix : <foo://rooms#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
select ?A1
from <file:rooms.n3>
where { { bind('2BedsInGuestroom' AS ?A1) . :guestroom a :RoomWithTwoBeds }
  UNION { bind('NoChairInGuestroom' AS ?A1) . :guestroom a :RoomWithoutChair }
  UNION { bind('c1InBedroom' AS ?A1) . { :c1 :in :bedroom }}
  UNION { bind('maybeC1InBedroom' AS ?A1) .
          FILTER NOT EXISTS { :c1 :in :bedroom }}}
```

[Filename: RDF/rooms2.sparql]

9.11 Rules in DL: Hybrid Reasoning

- Early Approaches: Donini, Lenzerini et al 1991; Levy, Rousset 1996 (CARIN): rather disappointing safety and decidability results: roughly, due to objects implicitly (existentially) assured by DL specifications.
- Newer investigations in Semantic Web context: DLV (Eiter et al 2004), DL+log (Rosati 2006); Motik, Sattler, Studer 2005; Lukasiewicz 2007: more detailed syntactical and structural constraints.
- SWRL (Semantic Web Rule Language; 2004):
  - Full Power of OWL-DL, allows for specifying undecidable settings, high computational complexity,
  - building upon the basic RULE-ML ontology for describing rules (rule; head, body; different kinds of atoms),
  - DL-safe rules (decidable) supported by Pellet: restriction in syntax and in semantics variables only applied to named resources (prunes the tableau; roughly ignoring all only existentially known objects).
- recall that SPARQL also returns only answers bound to explicitly known nodes (cf. Slide 417).

Simple Rule Example: Uncle

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla#>.
:sue :hasChild :barbara; :hasSibling :john.
:john :name "John"; :hasChild :alice, :bob;  :hasSibling :sue.
:x a swrl:Variable.
:y a swrl:Variable.
:z a swrl:Variable.
 :uncleAuntRule a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :hasUncleAunt;
                swrl:argument1 :x ;   swrl:argument2 :z ]);
   swrl:body ([ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :hasChild;
                swrl:argument1 :y ;   swrl:argument2 :x ]
              [ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :hasSibling;
                swrl:argument1 :y ;   swrl:argument2 :z ]).
```

[Filename: RDF/uncle-rule-swrl.n3]

```
prefix : <foo://bla#>
select ?X ?U
from <file:uncle-rule-swrl.n3>
where {?X :hasUncleAunt ?U}
```

[Filename: RDF/uncle-rule-swrl.sparql]

DL-SAFE RULES CONSIDER ONLY NAMED RESOURCES

- analogous to SPARQL queries and owl:hasKey (cf. Slide 404)
- only positive atoms in the body (then OWA vs. CWA does not play any role)

$\Rightarrow$ work only on a finite instantiated subgraph of the whole DL model

$\Rightarrow$ does not interfere with the blocking, and

$\Rightarrow$ does not break decidability.

Comparison: SWRL Rule, Property Chain, SPARQL, DL

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla#>.
:john :hasChild :bob; :hasSibling :paul, [].
:hasSibling a owl:SymmetricProperty.
:paul a [a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].

:uncleRule a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :uncle1;
                swrl:argument1 :y ; swrl:argument2 :z ]);
   swrl:body ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :hasChild;
                swrl:argument1 :x ; swrl:argument2 :y ]
              [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :hasSibling;
                swrl:argument1 :x ; swrl:argument2 :z ]).
   :x a swrl:Variable.    :y a swrl:Variable.   :z a swrl:Variable.
[ owl:propertyChain ([owl:inverseOf :hasChild] :hasSibling) ] rdfs:subPropertyOf :uncle2.
:Uncle owl:equivalentClass [a owl:Restriction; owl:onProperty :hasSibling;
  owl:someValuesFrom [a owl:Restriction; owl:onProperty :hasChild;
                      owl:minCardinality 1]].
```

[Filename: RDF/uncle-comparison.n3]

DL-Safe Rules consider only named Resources (cont'd)

```
prefix : <foo://bla#>
select ?N ?U1 ?U2 ?isU
from <file:uncle-comparison.n3>
where {{?N :uncle1 ?U1} union {?N :uncle2 ?U2}
       union {?isU a :Uncle}}
```

[Filename: RDF/uncle-comparison.sparql]

- blank nodes are considered. Paul and a bnode (John's other brother) are Bob's uncles.
- implicitly known nodes are not considered: John's brother Paul has a child (so John is also an uncle) which is only implicitly known.

BUILT-IN SWRL ATOMS

SWRL provides some built-in atoms for owl:sameAs, owl:differentFrom etc.

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla#> .

:name a owl:FunctionalProperty; a owl:DatatypeProperty.
:john a :Person; :name "John"; :hasChild :alice, :bob.
:alice a :Person; :name "Alice".
:bob a :Person; :name "Bob".
  :x a swrl:Variable.   :y a swrl:Variable.  :z a swrl:Variable.
:r a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :hasSibling;
                swrl:argument1 :y ; swrl:argument2 :z ]);
   swrl:body ([ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :hasChild;
                swrl:argument1 :x ; swrl:argument2 :y ]
              [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate :hasChild;
                swrl:argument1 :x ; swrl:argument2 :z ]
              [ a swrl:DifferentIndividualsAtom;
                swrl:argument1 :y ; swrl:argument2 :z ]).
```

[Filename: RDF/sibling-rule.n3]

Built-In SWRL atoms (Cont'd)

```
prefix : <foo://bla#>
select ?X ?CH ?SIB
from <file:sibling-rule.n3>
where {{?X :hasChild ?CH} union {?X :hasSibling ?SIB}}
```

[Filename: RDF/sibling.sparql]

EVALUATION OF DL REASONING VS SWRL RULES

- DL Reasoning considers implicitly known resources (= graph nodes) and handles them in the tableau via blocking:
  - Structurally identical graph fragments are not further explored. Due to DL's locality principle and tree structure of the model, the model can be kept finite.
- Rules are also incorporated into the tableau, but since they do not have the tree property, blocking would not be sufficient for keeping the model finite when implicitly known resources are considered.

$\Rightarrow$ if something "important" about an implicitly known node can only be derived by a rule, it is not discovered (cf. next example).

DL-SAFETY: SWRL RULES DO NOT CONSIDER IMPLICIT RESOURCES

(see Turtle fragment next slide)

- Rule: all persons believe in God,
- jack has a blank node child _:b0 who is a parent,
- _:b0 is a believer (by the rule),
- as the grandchild is a person, application of the rule would result in the fact that it believes in God, i.e. it is a believer, which should make _:b0 a ParentOfBeliever.
- how to show that the grandchild is not considered by the rule: add a statement that _:b0 is not parent of any believer.
- run “classify” for the n3:
  - the ontology is consistent,
  - _:b0 is accepted to be a :NotParentOfBeliever.

SWRL Rules and DL-Safety: Example

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla#> .
:jack a :Person; :hasChild [a :Person; a :Parent; a :NotParentOfBeliever].
:Parent owl:equivalentClass
  [a owl:Restriction; owl:onProperty :hasChild; owl:someValuesFrom :Person].
:Believer owl:equivalentClass
  [a owl:Restriction; owl:onProperty :believes; owl:minCardinality 1].
:ParentOfBeliever owl:equivalentClass
  [a owl:Restriction; owl:onProperty :hasChild; owl:someValuesFrom :Believer].
:NotParentOfBeliever owl:equivalentClass [a owl:Restriction;
     owl:onProperty :hasChild; owl:onClass :Believer; owl:cardinality 0].

:x a swrl:Variable.
:r a swrl:Imp;
   swrl:head ([ a swrl:IndividualPropertyAtom;
                swrl:propertyPredicate :believes;
                swrl:argument1 :x ; swrl:argument2 :god ]);
   swrl:body ([ a swrl:ClassAtom; swrl:classPredicate :Person;
                swrl:argument1 :x ]).
```

[Filename: RDF/hidden-prop.n3]

SWRL Rules and DL-Safety: Example (Cont'd)

```
prefix : <foo://bla#>
select ?B ?PB ?NPB
from <file:hidden-prop.n3>
where {{?B a :Believer} union {?PB a :ParentOfBeliever}
                         union {?NPB a :NotParentOfBeliever}}
```

[Filename: RDF/hidden-prop.sparql]

RULE EXAMPLE: BIG CITIES

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

mon:population  rdfs:range xsd:int; a owl:FunctionalProperty. ## all cities are different.
_:Million a rdfs:Datatype; owl:onDatatype xsd:int; owl:withRestrictions (_:m1).
_:m1 xsd:minInclusive 1000000 .

:ProvinceWithBigCity a owl:Class.   # otherwise sparql answer empty.
:ProvinceWithTwoBigCities a owl:Class.   # otherwise sparql answer empty.
:CountryWithTwoBigCities a owl:Class.   # otherwise sparql answer empty.

:HasBigPopulation owl:equivalentClass [a owl:Restriction;
  owl:onProperty mon:population; owl:someValuesFrom _:Million].
:BigCity owl:intersectionOf (mon:City :HasBigPopulation).
```

[Filename: RDF/bigcities-base.n3]

First: the simplest rule: a country where no provinces are contained in the database:

$$\text{Cw2BCs}(X) :- \text{Country}(X) \wedge \text{BigCity}(Y) \wedge \text{BigCity}(Z) \wedge \text{hasCity}(X, Y) \wedge \text{hasCity}(X, Z) \wedge Y \neq Z.$$

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

:CountryNoProvsRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :CountryWithTwoBigCities;
              swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Country; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :y ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :z ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :z ]
            [ a swrl:DifferentIndividualsAtom; swrl:argument1 :y ; swrl:argument2 :z ]).
```

[Filename: RDF/bigcities-country-noprovs-rule.n3]

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

:x a swrl:Variable.   :y a swrl:Variable.     :z a swrl:Variable.
:ProvBigCityRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithBigCity; swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Province; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :y ]).

:TwoProvsRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :CountryWithTwoBigCities; swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Country; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithBigCity; swrl:argument1 :y ]
            [ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithBigCity; swrl:argument1 :z ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasProvince;
              swrl:argument1 :x ; swrl:argument2 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasProvince;
              swrl:argument1 :x ; swrl:argument2 :z ]
            [ a swrl:DifferentIndividualsAtom; swrl:argument1 :y ; swrl:argument2 :z ]).
```

[Filename: RDF/bigcities-2provs-rules.n3]

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix mon: <http://www.semwebtech.org/mondial/10/meta#>.
@prefix swrl: <http://www.w3.org/2003/11/swrl#>.
@prefix : <foo://bla/>.

:Prov2BigCitiesRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithTwoBigCities; swrl:argument1 :x]);
 swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Province; swrl:argument1 :x ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :y ]
            [ a swrl:ClassAtom; swrl:classPredicate :BigCity; swrl:argument1 :z ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :y ]
            [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasCity;
              swrl:argument1 :x ; swrl:argument2 :z ]
            [ a swrl:DifferentIndividualsAtom; swrl:argument1 :y ; swrl:argument2 :z ]).

:Prov2CRule a swrl:Imp;
 swrl:head ([ a swrl:ClassAtom; swrl:classPredicate :CountryWithTwoBigCities; swrl:argument1 :x]);
   swrl:body ([ a swrl:ClassAtom; swrl:classPredicate mon:Country; swrl:argument1 :x ]
              [ a swrl:ClassAtom; swrl:classPredicate :ProvinceWithTwoBigCities; swrl:argument1 :y ]
              [ a swrl:IndividualPropertyAtom; swrl:propertyPredicate mon:hasProvince;
                swrl:argument1 :x ; swrl:argument2 :y ]).
```

[Filename: RDF/bigcities-prov-2bigcities-rules.n3]

Rule Example: Big Cities (Cont'd)

```
prefix : <foo://bla/>
prefix mon: <http://www.semwebtech.org/mondial/10/meta#>
select ?C ?BC ?P1 ?P2 ?X
from <file:bigcities-base.n3>
from <file:bigcities-2provs-rules.n3>
from <file:bigcities-prov-2bigcities-rules.n3>
from <file:bigcities-country-noprovs-rule.n3>
#from <file:dummy-cities.n3>     ## a small test setting
from <file:mondial-europe.n3>    ## europe is more than sufficient =:(
from <file:mondial-meta.n3>
where {# {?BC a :BigCity} UNION
       {?X a mon:Country; mon:carCode ?C; mon:hasCity ?BC . ?BC a :BigCity} UNION
       {?P1 a :ProvinceWithBigCity} UNION
       {?P2 a :ProvinceWithTwoBigCities} UNION
       {?X a :CountryWithTwoBigCities}}
```

[Filename: RDF/bigcities-by-rules.sparql]

Chapter 10
Jena: an API for Semantic Web Applications

There exist different APIs:

- SPARQL (discussed above, supported my most APIs)
- Jena API – RDF/RDFS/OWL in Java
- RDF4J API – RDF/RDFS/OWL in Java
- OWL API – Ontology Editing
- Hermit API – Reasoner
- Protégé – Graphical Tool for Ontology management
- Virtuoso – Web Server for Semantic Web Data

Jena: Overview

- Apache Jena is "a free and open source Java framework for building Semantic Web and Linked Data applications"
- Originally developed by HP Labs (until October 2009) since then at Apache.
- Apache Jena consists of:
  - RDF API : read and manipulate RDF graphs
  - ARQ : query RDF data with SPARQL up to SPARQL 1.1 and support for federated queries
  - Ontology API : supports the usage of RDFS and OWL to add semantics to the RDF data
  - Inference API : reasoner support with inference rules and in-built or external RDFS/OWL reasoners
  - TDB : a native triple store
  - Fuseki : a Web server to expose the RDF data over HTTP (SPARQL; also LOD??)
- An official Jena tutorial can be found at
  `https://jena.apache.org/tutorials/index.html`

Overview: Jena Model vs. InfModel/OntModel with Reasoning

Jena Model, InfModel, OntModel: Overview

- Model: the pure (RDF) graph.
  - Nodes, (Triple) Statements, RDF-Statements + reified Statements,
  - model operations: read, remove, add, union, difference, intersect, write
  - triple patterns, method: listStatements(s,p,o) [s,p,o constants or null]
- InfModel: (RDF) graph with any kind of inference and triple patterns
- OntModel (extends InfModel)
  - "understanding" of DL/OWL-level notions like Individual, OntClass, OntProperty; operations to create and handle complex class definitions (e.g., owl:Restrictions ...) (ontology generation, not necessarily with reasoning)
    - Lots of methods around the ontology metadata, like e.g. with an OntProperty p: Iterator<Restriction> i = p.listReferringRestrictions(); to obtain all owl:restrictions that deal with p.
    - `https://jena.apache.org/documentation/ontology/index.html`
  - optional: with DL reasoning support

10.1 The Jena RDF API (everything around Model/Graph)

- The Jena framework gives access to various objects that are linked to each other for managing and manipulating RDF data:
  - Dataset : Contains one default model and zero or more named models
  - Model : A collection of RDF statements. Technically a wrapper around a *Graph*, which gives access to a lot of convenience methods
  - Statement : Represents a RDF statement (S,P,O). Technically a wrapper around a *Triple*, which gives access to a lot of convenience methods
  - Resource, Literal, Property, RDFNode, ... : the parts of a Statement
- Example code: Java/JenaModel.java

First, generate a simple RDF Graph “Model”

- empty graph:
  Model family = ModelFactory.createDefaultModel();
  from a file:
  Model europe = RDFDataMgr.loadModel(“/home/Mondial/mondial-europe.n3”);
- Add data from a file or another model:
  model.add(Model *othermodel*);
  model.add(RDFDataMgr.loadModel(*filepath+name*));
  RDFDataMgr.read(model, *filepath+name*);

  The RDF serialization format is automatically chosen from the file extension/automatically detected by the parser (?) or can be specified explicitly in the call.
- use the direct Java commands to manipulate the model element-wise.
- compute model as union, intersection etc from other models.

Operations with Models

- Define prefixes

  europe.setNsPrefix(“mon”, “http://www.semwebtech.org/mondial/10/meta#”);
  family.setNsPrefix(“”, “foo://bla/meta#”);

  – is e.g. used when outputting the model in Turtle format.

- Output the Model in different RDF formats

  family.write(System.out, “TURTLE”);
  family.write(System.out, “RDF/XML-ABBREV”);

- Model operations (as sets of edges; vertices are only removed if *all* edges of them are removed)

  Model mUnion = model.union(model2);
  Model mIntersection = model.intersection(model2);
  Model mDifference = model.difference(model2);

Creation of Resources and Properties

- Resource and Property are interfaces (Property extends Resource)
  - later: with OntModel, all kinds of owl:Class'es are also Resources
- model.createResource(String uri) → Resource (uri = null → blank node)
  model.getResource(String uri) → Resource returns that resource *if it already exists*)
- model.createProperty(String namespace, String localname) → Property
  model.getProperty(String uri, String localname) → Resource returns that resource *if it already exists*)
  Properties like RDF.type, RDFS.label are predefined in
  `org.apache.jena.vocabulary.RDFS` etc.
- model.createLiteral(String value, String langtag) → language-tagged Literal
- res.getLocalName(), prop.getLocalName() → String,
  res.getNameSpace(), prop.getNameSpace() → String,
  ⇒ namespaces are just strings, there is no usage of prefixes here.

Adding Statements to the Graph

- resource.addProperty(Property p, RDFnode resource)
  resource.addProperty(Property p, String stringvalue)
  resource.addProperty(Property p, string lexical-form, RDFDatatype datatype)
  (Jena documentation: class XSDDatatype implements RDFDatatype)
  resource.addLiteral(Property p, float/double/long val)
  resource.hasProperty(...)/hasLiteral(...) → boolean for analogous tests
  resource.getProperty(Property p) → Statement returns *some* such statement
  resource.getPropertyResourceValue(Property p) → Resource gives *some* such resource
  resource.removeAll(Property p) → Resource removes all such statements

```
Resource john = family.createResource("foo://bla/persons/john")
  .addProperty(nameP, "John")
  .addProperty(nameP, "Johannes", "de")
  .addProperty(nameP, family.createLiteral("Jean", "fr"))
  .addProperty(family.createProperty("foo://bla/meta#","birthdate"),
               "1970-12-31", XSDDatatype.XSDdateTime)
  .addProperty(family.createProperty("foo://bla/meta#","child"),
               family.createResource()
                 .addProperty(nameP, "Alice")
                 .addLiteral(family.createProperty("foo://bla/meta#","age"), 10));
Resource alice = john.getPropertyResourceValue(family.getProperty("foo://bla/meta#","child"));
```

Access Data in the Model

- Get all property statements of a specific resource

```
Resource andorra = europe.getResource("http://www.semwebtech.org/"
                                      + "mondial/10/countries/AND/");
StmtIterator andIter = andorra.listProperties();
while (andIter.hasNext()) {
    Statement stmt      = andIter.nextStatement();  // get next statement
    Resource  subject   = stmt.getSubject();     // get the subject
    Property  predicate = stmt.getPredicate();   // get the predicate
    RDFNode   object    = stmt.getObject();      // get the object
         // object might be Resource or Literal -> superclass RDFNode
    //...
}
```

- res.listProperties(*prop*) $\rightarrow$ StmtIterator lists all statements for a given property.
- for handling the object position, use
  rdfnode.asResource() to "cast" it as jena.Resource (if it is one),
  rdfnode.asLiteral() to "cast" it as jena.Literal (if it is one),
  - literal.getValue() $\rightarrow$ some of the Java classes String, Float, Integer etc.,
  - literal.getFloat(), literal.getInt(), literal.getLanguage() etc.

Access data in the model: all Statements

- Iterate over all statements

```
StmtIterator iter = model.listStatements();
```

... an iterator over all statements in the model:

```
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement();  // get next statement
    System.out.println(stmt.toString());
}
```

ListStatements with Pattern

- Query the Model/Dataset without the ARQ component
- More programming style instead of query language
- Pattern matching: `model.listStatements(Subject, Predicate, Object)`
- Preset or null s/p/o to match a triple pattern,
- if object is null, it must be casted (because RDFNode or Literal (value) is allowed).

  Select all statements with 'hasCity' property that are located in France:

```
StmtIterator fcities = europe.listStatements(
  europe.getResource("http://www.semwebtech.org/mondial/10/countries/F/"),
  europe.getProperty("http://www.semwebtech.org/mondial/10/meta#hasCity"),
  (RDFNode) null);
```

  Select all things that have name "Monaco":

```
StmtIterator monacos = europe.listStatements(
  null,
  europe.getProperty("http://www.semwebtech.org/mondial/10/meta#name"),
  "Monaco");
```

ListStatements with an ExtendedIterator

- `StmtIterator`s (and NodeIterator, ResIterator etc) can use the interface `ExtendedIterator` that provides the following Methods:
- filterDrop(Predicate<T> f),
- filterKeep(Predicate<T> f)

Example to do

ListStatements with a Selector

Define a `Selector` that further filters by its `selects(Statement s)` function:

Return all things that have a population of more than 1,000,000:

```
StmtIterator bigthings = europe.listStatements(
   new SimpleSelector(null, europe.getProperty("http://www.semwebtech.org/"
                      + "mondial/10/meta#population"), (RDFNode) null) {
     public boolean selects(Statement s){
          return s.getObject().asLiteral().getFloat() > 1000000;
     }
   }
);
```

- note: the object position for “population” is guaranteed to be a numeric literal.
- filtering it to cities is only possible when using the dirty way via testing for the URI substring “/cities/”.

All resources with a certain property

Predefined namespaces in Jena

- Class resources cannot have user-defined properties except annotation properties like rdfs:label
- example: labels with language tags

```
Resource country = europe.getResource("http://www.semwebtech.org/"
                              + "mondial/10/meta#Country");
country.addProperty(RDFS.label, europe.createLiteral("Country", "en"));
country.addProperty(RDFS.label, europe.createLiteral("Land", "de"));
//Get all resources with rdfs:label
ResIterator tagIter = model.listResourcesWithProperty(RDFS.label);
while(tagIter.hasNext()){
  Resource res = tagIter.nextResource();
  StmtIterator prop = res.listProperties(RDFS.label);
  while(prop.hasNext()){
    Statement stmt = prop.nextStatement();
    System.out.print(stmt.getObject().asLiteral().getValue() + " ");
    System.out.println(stmt.getObject().asLiteral().getLanguage());
} }
```

yes/no checks for Properties of Resources

- *res*.hasProperty(*prop*) – (res, prop, X)?
  res.hasProperty(*prop*, *rdfnode*) – (res, prop, obj)?
  res.hasProperty(*prop*, *string*), *res*.hasProperty(*prop*, *string*, *langtag*),
  res.hasLiteral(*prop*, *int/long/float/double*)

Operations for Modifying and Deleting Statements

- *res*.removeAll(*prop*): remove all statements (*res, prop, X*)
- *res*.removeProperties(): remove all statements (*res, P, X*)
- *stmt*.changeObject(*rdfnode*),
  stmt.changeObject(*string*), *stmt*.changeObject(*string*, *langtag*),
- *stmt*.changeLiteralObject(*int/long/float/double*),
- *stmt*.remove(),
- *stmt*.createReifiedStatement() creates an rdf:Statement object with rdf:subject/rdf:predicate/rdf:object triples,
- *stmt*.removeReification(),
- ... and many more, see documentation for all these classes.

Datasets

- a Dataset contains a default model (maybe empty) and zero or more named graphs/models

```
Dataset ds = DatasetFactory.create();
ds.setDefaultModel(europe);
alice.addProperty(family.createProperty("foo://bla/meta#","livesIn"),
                  andorra);
    // alice is the alice object from the 'family' model,
    // andorra is the andorra object in the 'europe' model,
    // the triple is added to 'family' (to alice).
ds.addNamedModel("http://family", family);
```

  - names of named graphs in such a dataset are usually not he filenames (like in SPARQL queries with FROM NAMED); here the name URI of the named graph is set to “http://family”.
  - the additional statement is only contained in the dataset ds, if it is added to the model *before* the model is added to the dataset ds.
    $\Rightarrow$ the model is copied, not linked.

... this dataset will be queried on Slide 566.

10.2 Jena ARQ API

ARQ component: functionality to query Models/Datasets with SPARQL

Sample Code: Java/JenaModel.java

BASICS OF QUERYING

Define a Query object

- `QueryFactory` parses a query String to a `Query` (which then contains the parsetree)

```
String qs =
    "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +
    "prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
    "SELECT ?CN " +
    "WHERE { ?C a mon:Country . ?C mon:name ?CN } " +
    "ORDER BY ?CN ";

Query q = QueryFactory.create(qs);
```

Create a QueryExecution object for the query and choose the input data

The QueryExecution contains the algebra expression/query plan.

- `QueryExecutionFactory` creates the query plan and creates a `QueryExecution`:
  - if an existing Model is used, the query *must not* contain any FROM clause:

    ```
    QueryExecution qe = QueryExecutionFactory.create(q, europe);
    ```

  - if a Dataset (containing a default model (maybe empty) and zero or more named graphs/models) us used, the query *must not* contain any FROM clause:

    ```
    QueryExecution qe = QueryExecutionFactory.create(q, ds);
    ```

    (this will create a temporary model/dataset during evaluation which is removed afterwards)
  - if the query contains a FROM clause, a model/dataset *can not be given*:

    ```
    QueryExecution qe = QueryExecutionFactory.create(q);
    ```

  - Note: it is *not possible* to have a model/data set, *and* any FROM clause in the query. Then, the FROM clauses are *ignored*.

Execute the query

- `QueryExecution` provides different execution functions for the different query types (Ask, Select, Construct, Describe)

```
ResultSet res = qe.execSelect();
Model constrModel = qe.execConstruct(); // set of triples
```

- `ResultSet` acts like an iterator which consists of `QuerySolutions`

```
while (results.hasNext()) {
  QuerySolution sol = res.next();
  RDFNode n = sol.get("CN"); // access by variable names
}
```

- Don't forget to close the `QueryExecution` to free up resources!

```
qe.close();
```

Handling of ResultSets

- A `ResultSet` can be only iterated (incl. printed) once (this runs the nested loops etc. for the actual evaluation)!
- The original `ResultSet` is bound to the `QueryExecution`, so it is no longer valid once the `QueryExecution` is closed!

⇒ Materialize the `ResultSet` into a `ResultSetRewindable` to iterate it multiple times and to use it after the `QueryExecution` is closed

- `ResultSetFormatter` can be used to output `ResultSets` in different formats (Table, CSV, XML, JSON, ...)

```
ResultSetRewindable persistentRes = ResultSetFactory.copyResults(res);
qe.close();
int numberOfResults = ResultSetFormatter.toList(persistentRes).size();
System.out.println("Number of Results: " + numberOfResults);
persistentRes.reset();   // to use it again
while (persistentRes.hasNext()) {
  QuerySolution sol = persistentRes.next();
  RDFNode n = sol.get("CN");    // and do something with n
  System.out.println(n); }
```

Try-with-resource

- The “try-with-resource” (since Java 7) statement automatically closes resources that are declared in it
  (which has been done before with “finally { . . . ...}”)
- The resource must implement java.lang.AutoClosable (includes all objects which implement java.io.Closable)

```
try(QueryExecution qe = QueryExecutionFactory.create(q, model)) {
    ResultSetFormatter.out(qe.execSelect());
}
```

Example: query with a Dataset that combines two models

- consider the dataset from Slide 560.

```
String qs1 =
  "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +
  "prefix : <foo://bla/meta#>\n " +
  "SELECT ?C ?CN ?PN ?A " +
  "WHERE { "
   + "?C a mon:Country . ?C mon:name ?CN . "
   + "   OPTIONAL { ?P :livesIn ?CC }"
   + "GRAPH <http://family> { ?P :name ?PN ."
   + "      OPTIONAL { ?P :livesIn ?C; :age ?A }}}";

Query q1 = QueryFactory.create(qs1);
QueryExecution qe1 = QueryExecutionFactory.create(q1, ds);
ResultSet res1 = qe1.execSelect();
// do something with res1, e.g.
ResultSetFormatter.out(System.out, res1, q1);
qe1.close();
```

Example: query with FROM

- like in standalone-SPARQL, then use the result
- Example: extend the europe model with all islands (and their data) in seas that are mentioned in mondial-europe.n3 (those that do not belong to european countries):

```
String qs2 = " prefix : <http://www.semwebtech.org/mondial/10/meta#> " +
    " CONSTRUCT { ?I ?P ?O . ?O a ?C ; :name ?ON } " +
    " FROM <file:" + filepath + "mondial.n3> " +
    " FROM NAMED <file:" + filepath + "mondial-europe.n3> " +
    " WHERE { GRAPH <file:" + filepath + "mondial-europe.n3>  " +
    "          { ?S a :Sea; :locatedIn ?CO. ?CO a :Country . " +
    "            FILTER NOT EXISTS { ?CO :name 'Russia' }  } " +
    "        ?I :locatedInWater ?S . " +
    "        ?I ?P ?O . FILTER (!isBlank(?O)) " +
    "        OPTIONAL { ?O a ?C ; :name ?ON } " +
    "        FILTER NOT EXISTS  " +
    "          { GRAPH <file:" + filepath + "mondial-europe.n3> { ?I a :Island }}} ";
Query q2 = QueryFactory.create(qs2);
QueryExecution qe2 = QueryExecutionFactory.create(q2);
Model islands = qe2.execConstruct();
islands.write(System.out,"Turtle");
Model europeAndIslands = europe.union(islands);   qe2.close();
```

Remote queries

- Query remote SPARQL endpoints over HTTP
- similar functionality as the SERVICE keyword:

```
String qs =   // country names as before ...
Query q = QueryFactory.create(qs);    // as before
QueryEngineHTTP remoteQE = (QueryEngineHTTP) QueryExecutionFactory
    .sparqlService("http://www.semwebtech.org/mondial/10/sparql", q);
//remoteQE.setSelectContentType(WebContent.contentTypeResultsXML);
ResultSet res4 = remoteQE.execSelect();
ResultSetFormatter.out(System.out, res4, q);
remoteQE.close();
```

Prepared Statements

- ... similar to PreparedStatements in JDBC, but ... different:
- create the Query parsetree from the String,
- add a QuerySolutionMap as initial bindings
- then create the query plan, and execute it:

```
String qs =  // country names as before ...
    "SELECT ?CN " +
    "WHERE { ?C a mon:Country . ?C mon:name ?CN } ";
Query q = QueryFactory.create(qs);
QuerySolutionMap initialBinding = new QuerySolutionMap();
initialBinding.add("C", mondial.getResource("http://www.semwebtech.org/"
    + "mondial/10/countries/USA/"));
QueryExecution qe3 = QueryExecutionFactory.create(q, europeAndIslands,
    initialBinding);
ResultSet res3 = qe.execSelect();  // ... and process it ...
qe3.close();
```

Prepared Statements in Jena: Discussion

- Allow user input (values) and protect from injection attacks:
  Do not copy the user input into the string and compile it:

  ```
  SELECT * FROM country where code= '$input'
  ```

  user inputs "`D'; drop table city;`" and the query becomes

  ```
  SELECT * FROM country where code= 'D'; drop table city;
  ```

  $\Rightarrow$ user input is only handled as values.

- in JDBC, the query plan is *prepared/precompiled*, and the values are just submitted to the query plan. For multiple executions, the query plan is only generated once.

- in Jena, the query parsetree is pre-generated and can be reused, but the query plan is newly generated for each execution
  - less efficient, of the query plan is always the same
  - might be more efficient, if different bindings result in different optimal query plans

- the setting is different from SQL: in SQL, the "?" *must be set*, in SPARQL, all answers would be returned.
  - with a partial tuple of initial bindings, the sets of answer bindings can be constrained (like a single VALUES tuple).

10.3 Jena OntModel and InfModel

Note: this chapter does not deal with details of the reasoning (see pp. 603), but focuses on the Jena API for handling models.

Sample Code: Java/JenaOntModel.java

Architecture: Model, Ontology, Inference

- Ontology model $\leftrightarrow$ Reasoner $\leftrightarrow$ Union Graph (Base RDF graph + imported ontologies)

GRAPHICS TODO

- triples of the base RDF graph (ABox)
- 0...n imported ontologies (TBox knowledge, usually OWL-DL)
  recall: many DL terms are represented by multiple triples – here, not the triples are part of the graph, but the whole DL concept is part of the knowledge
- the chosen reasoner & reasoner mode
- triples that were derived through the reasoner are 'known" to the Ontology access and to the reasoner
  - they can be materialized and stored in the RDF graph or in a separate RDF graph (bottom-up style, often followed by rule-based reasoners)
  - they might be "virtual". Queries are then answered by a (tableau) reasoner. (Partial) materialization of such virtual conclusions (e.g. tabling) is possible.

Creation of an OntModel – Ontology Handling, optionally with Reasoning

- use a Jena OntModelSpec to distinguish the capabilities of the ontology (OWL, OWL-DL, OWL-Lite, RDFS) and the capabilities of the reasoner that are required:
  `http://jena.apache.org/documentation/javadoc/jena/org/apache/jena/ontology/OntModelSpec.html`
  - OWL_DL_MEM: OWL DL model stored in memory, *no reasoning*
  - OWL_DL_MEM_TRANS_INF: ... use the transitive inferencer for rdfs:subClassOf/subPropertyOf
  - OWL_DL_MEM_RDFS_INF: ... same, use the RDFS inferencer
  - OWL_DL_MEM_RULE_INF: ... same, using the Jena OWL Rule Inferencer
  - OWL_MEM_MINI_RULE_INF: ... some OWL
  - OWL_MEM_MICRO_RULE_INF: ... little bit of OWL
  - and some more ...
  - "alien" reasoners like Pellet/Openllet can also provide an own spec.

Creation of a Jena OntModel with a Jena Spec

- create an empty OntModel:
  `OntModel m = ModelFactory.createOntologyModel(OntModelSpec spec)`
- for given: `Model base = ...`:
  `OntModel m = ModelFactory.createOntologyModel(OntModelSpec spec, Model base)`
- initialize with an ontology from <uri>:
  `OntDocumentManager dm = OntDocumentManager.getInstance() OntModel m = dm.getOntology(String uri, OntModelSpec spec)`
- read data from another file into an existing OntModel:
  `ontmodel.read(`*`filepath+name`*`);`
- *ontmodel*.getReasoner() $\rightarrow$ gives the reasoner.

Creation of a Jena OntModel with Pellet

```
Model model = ...  initialization ... ;
OntModel pelletmodel =
  ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC);
pelletmodel.add(model);
// pelletmodel.prepare();   -- not needed
// Reasoner pellet = pelletmodel.getReasoner();  -- if needed
```

Usage of Jena's OntModel

- all methods from InfModel (see Slide 578)
- Classes/Interfaces (below called XXX) for each type of DL expressions:
  Individual, OntClass, Restriction (SomeValuesFromRestriction, ..., ComplementClass, IntersectionClass MaxCardinalityQRestriction, HasValueRestriction, ...), Property (DatatypeProperty, ObjectProperty, TransitiveProperty, ...)
- createXXX(*parameters*): constructors for things and expressions ... e.g.
  createIndividual(*string uri*), createClass(*string uri*),
  createMaxCardinalityQRestriction(*string uri, prop, int cardinality, OntClass cls*)
  createHasValueRestriction(*string uri, prop, RDFNode value*)
  createComplementClass(*string uri, Resource cls*)
- Ontology-level methods to access the model:
  - m.getXXX(*string uri*): accessor for things and expressions (to result in an instance of the correct Java class)
  - m.listXXX(...): return an iterator over all things of a certain type, listIndividuals(), listIndividuals(Resource cls), listNamedClasses(), listFunctionalProperties(), listRestrictions(), ...

Usage of Jena's OntModel (cont'd)

- write(...): export the *raw model* (base graph + explicit OWL stuff) *without inferred things* to a file or stream.
- writeAll(...): export all triples (base + inferred) to a file or stream

INFMODEL

- Pure InfModel: RDFS with a reasoner – methods for controlling the reasoning capabilities.
- OntModel extends InfModel with the previously described DL stuff.

Creation of an InfModel – Graph + Reasoner

- An RDFS Model is not an OntModel (it understands nothing than RDF/RDFS):
  `ModelFactory.createRDFSModel(Model model)`
- With RDF data and RDFS metadata from different Model instances:
  `ModelFactory.createRDFSModel(Model schema, Model model)`
- One can also use an OWL Reasoner with just an InfModel:
  `ModelFactory.createInfModel(Reasoner reasoner, Model model)`
  Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
  InfModel infmodel = ModelFactory.createInfModel(reasoner, model);
- `ModelFactory.createInfModel(Reasoner reasoner, Model schema, Model model)`

Usage of Jena's InfModel (and OntModel)

- `validate()` check consistency of the underlying data; returns a ValidityReport.
  - `boolean isValid()` Note: "inconsistent classes" i.e., classes that can be derived to be empty are only considered to be non-valid if also an instance is asserted.
  - `boolean isClean()` if there are inconsistencies *or* other warnings (e.g., inconsistent classes).
  - `Iterator<...> getReports()` for details
- `void prepare()` Perform any initial processing and caching.
  (usually done automatically when needed)
- `void rebind()` reconsult the underlying data to take into account changes that did not happen via the InfModel/OntModel interface to the underlying graph.
- `void reset()` Reset any internal caches (e.g. Tabling)

ONTMODEL: THE JENA ONTOLOGY API

- Ontology model: strict separation between individuals, classes, and properties:
  - Individuals have user-defined properties
  - classes and properties have only properties defined in RDFS/OWL, and they might have *owl:AnnotationProperties* (e.g. rdfs:label)
  - these properties might be queried and modified (which changes the reasoner's knowledge base)
- Individuals and classes are anonymous (blank nodes) or have a URI;
  - createClass(), createClass(*uri-string*), [more on classes corresponding to DL concepts see subsequent slides]
  - createIndividual(Resource *class*), createIndividual(String uri, Resource *class*), (where *class* might be owl:Thing),

Properties in the OntModel

- Properties must have a URI, and they must be created according to their specific characteristics:
  - createDatatypeProperty(*uri*), createDatatypeProperty(*uri*, boolean *isfunctional*), createObjectProperty(*uri*), createObjectProperty(*uri*, boolean *isfunctional*), More specific subclasses of ObjectProperties: createTransitiveProperty(*uri*), createSymmetricProperty(*uri*), createInverseFunctionalProperty(*uri*), createInverseFunctionalProperty(*uri*, boolean *isfunctional*)

DL Concept definitions in the OntModel

Each DL expression type has a constructor:

- TODO

- Add/remove class membership of some Individual:

```
indiv.addOntClass(Resource cls);  res.removeOntClass(Resource cls);
```

- Retrieve an `OntResource` and list all its properties

```
OntResource res = ontM.getOntResource("http://www.semwebtech.org/"
    + "mondial/10/country/AL");
for(StmtIterator props = res.listProperties(); props.hasNext(); ){
    Statement stmnt = props.next();
    System.out.println("property: " + stmnt.getPredicate()
        + " value: " + stmnt.getObject().toString());       }
```

Accessing Resources in an OntModel

- Access an `OntResource` as an `OntClass` and list all its superclasses

```
OntClass cls = res.as(OntClass.class);
//listSubClasses/SuperClasses also available with boolean argument
//for only directly inferred relationships
for(ExtendedIterator<OntClass> supers = cls.listSuperClasses();
    supers.hasNext(); ){
    OntClass superClass = supers.next();
    System.out.println(superClass + " is a superclass of " + cls.getURI()); }
```

- list all instances of an OntClass:

```
ExtendedIterator<? extends OntResource> listInstances()
```

  and many other methods ...

- Analogous: access an `OntResource` as an `OntProperty` and investigate/manipulate its properties.

AllDifferent in Jena

- (like many things in Jena) AllDifferent is an interface ... so it does not have a constructor, but things (here: an existing class, or an ad hoc set of instances) have to be “turned into” an AllDifferent:
- many such questions can be answered e.g. by google “jena usage alldifferent”
- Pseudo-“Cast” a class as AllDifferent, *and then add all its instances as different members*

```
Resource c = mondial.getResource("http://www.semwebtech.org/"
      + "mondial/10/meta#Country"));
OntClass oc = c.as(OntClass.class);
AllDifferent ac = c.asAllDifferent();
// add the instances to the AllDifferent specification list:
ac.addDistinctMembers(oc.listInstances);
```

EXAMPLE: CREATE COMPLEX CLASS EXPRESSIONS

Task: Define a class for all individuals that love at least one cat and also love to run.

- New empty `OntModel` and new prefix

```
OntModel animalModel = ModelFactory
    .createOntologyModel(OntModelSpec.OWL_DL_MEM);
String catPrefix = "http://www.semwebtech.org/cats/";
```

- Define classes and properties

```
OntClass catClass = animalModel
    .createClass(catPrefix + "meta#" + "Cat");
OntClass activityClass = animalModel
    .createClass(catPrefix + "meta#" + "Activity");
ObjectProperty lovesProp = animalModel
    .createObjectProperty(catPrefix + "meta#" + "loves");
```

- Define "running" as an individual of the Activity class

```
Individual run = activityClass.createIndividual(catPrefix + "running");
```

- Define a catLover class

```
SomeValuesFromRestriction lovesACat = animalModel
    .createSomeValuesFromRestriction(null, lovesProp, catClass);
```

- Define a runner class

```
HasValueRestriction lovesRunning = animalModel
    .createHasValueRestriction(null, lovesProp, run);
```

- Combine the two restrictions

```
UnionClass runningCatLoverClass = animalModel
    .createUnionClass(catPrefix + "meta#runningCatLover",
                      animalModel.createList(new RDFNode[]
                            {lovesACat, lovesRunning})
                      );
```

ENUMERATED CLASSES

- Defining a class by stating the individuals the class contains

```
String dogPrefix = "http://www.semwebtech.org/dogs/";
String humanPrefix = "http://www.semwebtech.org/humans/";
OntClass humanClass = animalModel
    .createClass(humanPrefix + "meta#Human");

EnumeratedClass dogLoverClass = animalModel
    .createEnumeratedClass(dogPrefix + "meta#dogLover", null);
dogLoverClass.addOneOf(humanClass.createIndividual(humanPrefix + "Basti"));
dogLoverClass.addOneOf(humanClass.createIndividual(humanPrefix + "Lars"));
```

Use auxiliary Ad-hoc-“query”-classes with InfModel (and OntModel)

- extension of Model.listStatements(s,p,o):

```
StmtIterator listStatements(Resource subject, Property predicate,
                            RDFNode object, Model posit)
```

  - Preset or null s/p/o to match a triple pattern,
  - if object is null, it must be casted:
    `(RDFNode) null`
  - “posit” is an additional separate model, where e.g. complex classes might be defined that are only used in a match pattern
    * typical case: match (null, rdf:type, *class*) where *class* is a complex ad hoc class definition.
    * Java Code next slide.
      · note: example uses simple OWL_DL_MEM spec without reasoning; no reasoning needed here,
      · Creates a class HasValue(neighbor,germany) in the posit model and asks for its members.

```
import org.apache.jena.rdf.model.Statement;
import org.apache.jena.rdf.model.StmtIterator;
import org.apache.jena.util.FileManager;
import org.apache.jena.vocabulary.RDF;

public class JenaListStmtsPosit {
  static String filepath = "/home/may/teaching/SemWeb/RDF/";
  public static void main(String[] args){
    String monPrefix = "http://www.semwebtech.org/mondial/10/";
    Model m = FileManager.get().loadModel(filepath + "mondial-europe.n3");
    OntModel ontM = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM, m);

    Model m2 = ModelFactory.createDefaultModel();
    OntModel posit = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM, m2);

    // create Class: HasValue(neighbor,germany) in posit and output its members
    OntClass Country = posit.createClass(monPrefix + "meta#Country");
    Individual germany = Country.createIndividual(monPrefix + "countries/D/");
    ObjectProperty neighborProp = posit.createObjectProperty(monPrefix + "meta#neighbor");
    HasValueRestriction GermanNeighbor =
       posit.createHasValueRestriction(null, neighborProp, germany);

    for (StmtIterator i =
         ontM.listStatements(null, RDF.type, GermanNeighbor, posit);
         i.hasNext(); ) {
           Statement s = i.nextStatement();
           Resource r = s.getSubject();
           System.out.println(r);
      }}}
```

[Filename: Java/JenaListStmtsPosit.java]

Aside: How does the Reasoner “find” and distinguish DL Specifications from input

- Recall Slide 391: owl:restriction Specifications that are not given using blank nodes are ignored
  $\Rightarrow$ when reading an input file, all triples forming such an expression must be “encapsulated” by a blank node (which then actually exists; cf. Slide 410)
- what happens if such a blank node is constructed with the Model API:

  TODO Java

General Notes for using Jena and Pellet/Openllet in this course

Three possibilities:

- the semweb.jar is an "uber-jar" which contains *all* classes from jena+pellet (and some more). Put it in the class path or the lib directory/build path – this is sufficient.
  - shell: call

    `javac -cp .:`*`path-to-semweb.jar`* `bla.java`

    `java -cp .:`*`path-to-semweb.jar`* `bla.java`

    (don't forget to have "." on the path for your own class)
  - note: such an "uber-jar" does not contain "nested" jars, but all jars (basically, these are zip-archives) are unpacked and re-packed with a manifest (pointing to the main class if existing)
- download Jena as jar files and put all in your classpath
- download and install Jena as maven project.
  - maven allows to specify dependencies wrt. exact versions of packages
  - automatically transitively downloads everything

10.4 Aside: Jena TDB API

- Basic Jena: a simple main-memory graph structure
- The Jena TDB component features a RDF quad store supporting the following:
  - quads for triples in optionally named graphs:
    (Subj, Pred, Obj, GraphID).
  - unchanged usage of the full range of Jena APIs
  - bulk loading
  - transactions
  - customizable query optimization
  - quad filtering
- Note: TDB2 is the newer version, but not compatible with TDB1 (which is obsolete)

TDB-based Datasets

- A `Dataset` instance contains the RDF data graphs (cf. Slide 560)
- Creating/loading a TDB2-backed dataset

```
Dataset dataset = TDB2Factory.createDataset(); //In-memory
Dataset dataset = TDB2Factory.connectDataset("./resource/TDB2test");
```

Administration

- TDB2 database folder structure
  - Backups/
  - Data-0001/ (with a lot of files in it that form the RDF store)

    ...

    tdb.lock
- the newest Data-000n is the *current* database where TDB works with
- compaction & backup of the database in Java:
  - `DatabaseMgr.compact(dataset.asDatasetGraph());`
    creates Data-000n+1 by reorganizing the current Data-000n, further accesses will then use the new one.
  - `DatabaseMgr.backup(dataset.asDatasetGraph());`

Constraints

- RDF store only (in a single directory) on a single machine, not distributed
- A TDB directory can only be accessed at the same time from a single JVM (but from multiple threads/processes/commandline calls; for transactionality see Slide 596)
  - Use the Fuseki server to make it available for multiple applications (Web server)

Command-line tools

Most of the following functionality is also available through the use of command-line scripts provided in the Jena binary distribution located in the `bin / bat` folders.

- `tdb2.tdbloader` , `tdb2.tdbquery` , `tdb2.tdbdump` , ...
- all with `-loc` *`unixpath`* argument to address the TDB2 directory on a computer
- (Note: `tdbloader2` is a TDB1 script and will destroy your TDB2 database)

Dataset in TDB

- Dataset storage
  - one directory is used to store one RDF dataset
  - unnamed graph of the dataset as a single graph
  - named graphs in a collection of quad indexes
  - All the named graphs can be treated as a single graph which is the union (RDF merge) of all the named graphs. This is given the special graph name <urn:x-arq:UnionGraph> in a GRAPH pattern. (default graph not included)
- Working with TDB Datasets
  - Normally, ARQ interprets FROM/FROM NAMED as coming from the web; the graphs are read using HTTP GET
  - TDB modifies this behaviour; instead of the universe of graphs being the web, the universe of graphs is the TDB data store

  $\rightarrow$ FROM and FROM NAMED describe a dataset with graphs drawn only from the TDB data store

Transactions

- Provides ACID (Atomicity, Consistency, Isolation, Durability) transaction support through the use of write-ahead-logging
  - All changes are written to journals, then propagated to the main database at a suitable moment
- Constraint: One active write transaction, multiple read transaction at the same time (globally to the whole TDB, not on the data-item level like in SQL database)
- Classical way of implementing transactions by BOT/EOT:

```
dataset.begin(ReadWrite.READ);
try{
    Model tdbM = dataset.getDefaultModel();
    ... read operation
} finally {dataset.end(); }

dataset.begin(ReadWrite.WRITE);
try{
    ... write operation
    dataset.commit();     //   dataset.abort()
} finally {dataset.end();}
```

TXN interface for Transactions

- Transactions using the TXN interface:

```
Txn.executeWrite(dataset, () -> {
        RDFDataMgr.read(dataset, "./resource/mondial-europe.n3");
        ... further commands that read and write the TDB contents ...
    });

Txn.executeRead(dataset, () -> {
        RDFDataMgr.write(System.out, dataset, Lang.TRIG);
        ... further commands that read the TDB contents ...
    });
```

  Note: variables that are communicated into the Lambda-function have to be final or effectively final.

- The TDB dataset can be used without these constructs, but then it is up to the user to implement the mutual exclusion/isolation
  - If the database becomes corrupted, it is not repairable
- No autocommit mode
- Models & graphs can be passed across transactions

TDB Query Optimization

The TDB optimizer utilizes both static and dynamic optimization to improve query execution

- Static optimization : transformations of the SPARQL algebra performed before query execution begins
- Dynamic optimization : deciding the best execution approach during the execution phase (may take into account data so far retrieved)
- Supported strategies: statistics-based, fixed, no reordering:
  TDB chooses a strategy depending which of the following files is present in the database directory:
  - none.opt : No reordering - execute triple patterns in the same order as in the query (still does filter placement)
  - fixed.opt : Built-in reordering based on the number of variables in a triple pattern
  - stats.opt : Content of the file are weighing rules for approximate matching counts of triple patterns
  - (none.opt & fixed.opt can be zero-length files)

Logging query execution – similar to SQL's query plan

- Messages send on "info" level
  - E.g. log4j in log4j.properties file:

    ```
    log4j.logger.org.apache.jena.arq.exec=INFO
    ```

- Fine-tuning output with Explain.InfoLevel
  - INFO : Log each query
  - FINE : Log each query and it's algebra form after optimization
  - ALL : Log query, algebra and every database access (can be expensive)
  - NONE : No information logged
- Activating the query logger:
  - globally: `TDB.getContext().set(ARQ.symLogExec,Explain.InfoLevel.FINE);`
  - single query:

    ```
    qExec.getContext().set(ARQ.symLogExec,Explain.InfoLevel.FINE);
    ```

  - commandline:

    ```
    tdbquery -loc=unixpath -set tdb:logExec=FINE -file queryfile
    ```

Internal Optimization: Filter placement (visible with the logger's query plan functionality)

- optimization of filtered basic graph patterns
- decides the best order of triple patterns in a basic graph pattern and also the best point at which to apply the filters within the triple patterns
- any filter expression of a basic graph pattern is placed immediately after all it's variables will be bound
- conjunctions at the top level in filter expressions are broken into their constituent pieces and placed separately.

```
SELECT ?c ?city
WHERE {
        ?c a mon:Country .
        ?c :hasCity ?city .
        ?city :population ?cpop .
        FILTER (?c = "<.../countries/D>"
                && ?cpop > 100000)
    }
```

=>

```
SELECT ?c ?city
WHERE {
        ?c a mon:Country .
        FILTER (?c="<.../countries/D>")
        ?c :hasCity ?city .
        ?city :population ?cpop .
        FILTER (?cpop > 100000)
    }
```

Statistics rule file

- Generating a statistics file:
  - commandline: `tdbstats -loc=DIR [-graph=URI]`
  - don't feed the output directly into the stats.opt
- File structure

```
(prefix ...
    (stats
        (meta ...)
        rule
        rule
    ))
```

- Rule structure

```
( (subj pred obj) count)
```

- Where `subj pred obj` are RDF terms or one of the following tokens:
  TERM, VAR, URI, LITERAL, BNODE, ANY

- Excerpt of stats.opt when using on mondial-europe

```
(stats
    (meta
        (timestamp "2019-05-09T16:37:00.929+02:00"
                        ^^<http://www.w3.org/2001/XMLSchema#dateTime>)
        (run@ "2019/05/09 16:37:00 MESZ")
        (count 62463))
    ((VAR <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.semwebtech.org/mondial/10/meta#River>) 185)
    ((VAR <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.semwebtech.org/mondial/10/meta#Country>) 54)
    ...
    (<http://www.semwebtech.org/mondial/10/meta#latitude> 1757)
    (<http://www.semwebtech.org/mondial/10/meta#isBorderOf> 180)
    (<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> 12123)
    ...
    (other 0))
```

Chapter 11
OWL Profiles, Rule-Based Reasoning, and Handling Reasoning with the Jena API

TYPES OF REASONING

- DL-Reasoner: Tableau Reasoning, FOL-based
- Rule-Based Reasoning (cf. “Deductive Databases” Lecture)
  - RDFS is purely rule-based
  - Inverse, Symmetry, and Transitivity are rule-based
  - Functionality (maxCard = 1) application for equating objects is rule-based
  - the above are all *purely positive rules*
  - Negation in the body: CWA (incl. stratification and WFS) vs. OWA
  - Disjunction in the head/choices
    (is also needed for cardinalities > 1 and some other things)
    $\Rightarrow$ Stable Models

DL/Tableaux vs. Deductive Databases/Datalog

- DL-Reasoner: Tableau Reasoning
  - can be extended easily with additional Tableau Rules
  - main problem: strategies, blocking, ...
  - strategies detect where exponential growth of the tableau occurs. Try to keep polynomial if possible.
- Rule-Based Reasoning (cf. “Deductive Databases” Lecture)
  - Prolog/Datalog (Resolution Calculus, “backward-chaining”)
  - Bottom-up database completion (“forward-chaining”, T_P^ω-operator, optimizations like “seminaive evaluation”, “magic sets”)
  - not restricted to special constructs (like DL/Tableaux), can handle rules in general (cf. SWRL)
  - Specific problems with negation (Closed-World), stratified/well-founded semantics
  - disjunction in the head: no classical rules, only via stable models (which is basically more related to Model Checking and Tableaux)
  - well-suited for ABox reasoning (data, databases), less well-suited for TBox reasoning (*theorem proving*, ontologies)

OWL PROFILES

- recall: OWL Variants:
  - OWL Lite (not explicitly discussed on the previous slides ...),
  - OWL-DL (equivalent to Description Logics),
  - OWL Full (syntax of RDF+RDFS+OWL, semantics undecidable, partially critical)
- From practical considerations, the *OWL profiles* are more important:
  `http://www.w3.org/TR/owl2-profiles/`
  - OWL2-EL: for ontologies that contain very large numbers of properties and/or classes. Basic reasoning problems can be performed in time that is polynomial *with respect to the size of the ontology* – not to the data
  - OWL2-QL: applications that use very large volumes of instance data, and where query answering is the most important reasoning task. Conjunctive query answering can be implemented using conventional relational database systems.
  - OWL-RL: scalable reasoning without sacrificing too much expressive power. OWL 2 RL reasoning systems can be implemented using rule-based reasoning engines.
  - note: none of the profiles is a subset of another

OWL2-RL

https://www.w3.org/TR/owl2-profiles/#OWL_2_RL

- OWL2-RL constrains the set of allowed constructs (e.g., no cardinalities other than 0,1,*), *and the usage* of constructs.
- An important restriction is the usage in subClassOf axioms:
  - Consider that subclass assertions are often used in RDFS:

    $\exists p.D \sqsubseteq C$ (domain) and $C \sqsubseteq \forall p.D$ (range).

    Typical OWL usage:

    $C \sqsubseteq \exists p.D$ to assert existential things, needs skolemization, can create infinite chains,

    $\forall p.D \sqsubseteq C$ check whether a universal restriction holds to define a class; needs to prove non-existence
  - OWL2-RL restriction:
    - $\exists p.D$ only on the subclass side,
    - $\forall p.D$ only on the superclass side,
    - compatible with RDFS, not with more expressive OWL use.

OWL2-RL (cont'd)

- Usage of Complement: only in superclass position *classexpr* $\sqsubseteq \neg D \rightarrow$ usage only as constraint to state what is not allowed
- Usage of Union and owl:oneOf only in subclass position:
  $(C_1 \sqcup C_2) \sqsubseteq D \quad \rightarrow \quad C_1 \sqsubseteq D$ and $C_2 \sqsubseteq D$, not in the head as true choice.
- usage of maxCardinality only 0,1 and only on superclass side (as restrictions)
- intersection and $\exists p.obj$ and $\exists p.value$ is allowed on both sides

$\Rightarrow$ does not cover everything of OWL Lite (= $\mathcal{SHIF}$, allows $\exists p.D$ without restriction) but also covers parts of OWL DL.

Translation to rules see `https://www.w3.org/TR/owl2-profiles/#OWL_2_RL`
(Entries prp-npa1, prp-npa2 seem to be wrong (require (?x a NegativeObject/DataPropertyAssertion)). For cls-svf-1 and csl-svf-2 consider that $\exists p.C$ is only allowed as subclass axiom $\exists p.C \sqsubseteq D$.)

- The internal rule-based reasoner of Jena goes beyond OWL-RL (and can run into nontermination, see Slide 617)

RULE-BASED REASONING

- cf. Datalog
- first-order: cannot reason about classes (predicate symbols), but only about individuals
- might apply second-order rule *patterns* e.g. for transitivity, subclass/subproperty, domain/range ("syntactically second-order", but actually only a first-order mechanism)
- OWL world: open-world negation, Rules: closed-world negation

  $\Rightarrow$ no way out in this issue (recall that well-founded semantics and stable models are also CWA)

  $\Rightarrow$ allow only positive rules
- For rule equivalents to the OWL-RL constructs allowed on OWL-RL see `https://www.w3.org/TR/owl2-profiles/#OWL_2_RL`.
  Choose those that are needed for a certain problem.
- Rule-based semantics has problems when allowing to derive existential objects: in OWL-RL *class [membership] assertions* (c a *class*), classes of the type $\exists p.C$ are not allowed.
- user-defined rules can be added.

11.1 Reasoning in Jena

https://jena.apache.org/documentation/inference/

Using an Ontology-Aware Reasoner

- can be used with the OntModel interface
- OntModel provides query/update methods on the level of Classes/Properties/DL-Concepts
- initialize the Model with a certain “OntModelSpec” (= complexity/expressiveness level) where Jena automatically employs and configures an internal OWL reasoner.
- can also be initialized with “foreign” reasoners that provide Jena adaptation (e.g. Pellet/Openllet)

Using another Reasoner

- can be used with the InfModel interface
- OntModel is a “subinterface” of InfModel, i.e. OntModel refines/extends InfModel
- combines a (=any) reasoner with an RDF model
  - used for “alien” external reasoners
  - used for Jena’s (non-ontology-aware) General Rule-Based Reasoner

USING PELLET AS REASONER

- Pellet is a tableau-based OWL-DL reasoner which can be used with an OntModel.

Example: Win-Move-Game: Draw Nodes

- Pure OWL cannot populate the DrawNode class
- use procedural programming to add triples (x a :DrawNode) to the model after reasoning
- note: it is not necessary to call prepareModel() explicitly
- Java Code next slide

```
import org.apache.jena.util.FileManager;
import openllet.jena.PelletReasonerFactory;

public class JenaPelletWinMoveDraw {
  static String filepath = "/home/may/teaching/SemWeb/RDF/";
  public static void main(String[] args){
    Model m = FileManager.get().loadModel(filepath + "winmove-axioms.n3");
    m.add(FileManager.get().loadModel(filepath + "winmove-closure.n3"));
    m.add(FileManager.get().loadModel(filepath + "winmove-graph.n3"));
    OntModel pelletmodel = ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC, m);

    String q = "prefix : <foo://bla#>" +
        "construct { ?N a :DrawNode }" +
        "where { ?N a :Node . filter not exists { ?N a :WinNode }" +
        "                     filter not exists { ?N a :LoseNode }}";
    Query qu = QueryFactory.create(q);
    QueryExecution qe = QueryExecutionFactory.create(qu, pelletmodel);
    Model resultgraph = qe.execConstruct();

    pelletmodel.add(resultgraph);

    q = "prefix : <foo://bla#>" +
        "select ?W ?L ?D " +
        "where {{ ?W a :WinNode } union { ?L a :LoseNode } union { ?D a :DrawNode }}";
    qu = QueryFactory.create(q);
    qe = QueryExecutionFactory.create(qu, pelletmodel);
    ResultSet results = qe.execSelect();
    ResultSetFormatter.out(results);
} }
```

[Filename: Java/JenaPelletWinMoveDraw.java]

Insights into derivations in Jena's InfModel (and OntModel)

- `Model getRawModel()`: the underlying RDF graph only.

Further methods are only partially (dependent on the reasoner) supported:

- `Model getDeductionsModel()`: The triples that are added to the base graph due to reasoning.

  Before this operation, all reasoning is evaluated first.
  It is not possible to see an "intermediate", partially on-demand state.

- if getDeductionsModel() does not correctly return all deductions:
  model.difference(model.getRawModel()).difference(model.getDeductionsModel())
  yields derived statements that are *not* in the Deductions Model ...

- `void setDerivationLogging(boolean logOn)`

  Switch on/off drivation logging. This can use a lot of space.

- `Iterator<Derivation> getDerivation(Statement stmt)`

- Derivation:
  - `printTrace(PrintWriter out, boolean bindings)`

- next slide: Java test code with pellet.

```
public class JenaPelletWinMove {
  static String filepath = "/home/may/teaching/SemWeb/RDF/";
  public static void main(String[] args){
    Model m = FileManager.get().loadModel(filepath + "winmove-axioms.n3");
    m.add(FileManager.get().loadModel(filepath + "winmove-closure.n3"));
    m.add(FileManager.get().loadModel(filepath + "winmove-graph.n3"));
    OntModel pelletmodel = ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC, m);
    Reasoner pellet = pelletmodel.getReasoner();
    pellet.setDerivationLogging(true);
    pelletmodel.prepare();

    Model dedModel = pelletmodel.getDeductionsModel();
    if (dedModel == null) System.out.println("DedModel is null");   // it is null ...
        else dedModel.write(System.out,"N3");
    Model rawModel = pelletmodel.getRawModel();
    Model diffModel = pelletmodel.difference(rawModel); // .difference(dedModel);
    System.out.println("---- DiffModel: ----------------------");
    diffModel.write(System.out,"N3");   // everything is in the DiffModel

    PrintWriter out = new PrintWriter(System.out);  // no derivations available
    for (StmtIterator i = diffModel.listStatements(null, RDF.type, (RDFNode) null); i.hasNext(); ) {
        Statement s = i.nextStatement();
        out.write("Statement is " + s + "\n");
        Iterator <Derivation> ds = pelletmodel.getDerivation(s);
        if (ds == null) System.out.println("DerivationsIterator is null");
        else for (Iterator<Derivation> i2 = pelletmodel.getDerivation(s); i2.hasNext(); ) {
            Derivation deriv = (Derivation) i2.next();
            deriv.printTrace(out, true);
    }   }
    out.flush();  } }
```

[Filename: Java/JenaPelletWinMove.java]

Code/Test Example: Pellet, Win-Move-Game

- getDeductionsModel() is null
- conclusions are in diffModel = pelletmodel.difference(rawModel);
- no derivation information available
  - Derivation information is especially useful, when an ontology is derived to be inconsistent

  $\Rightarrow$ pellet returns error messages that do not always give good insights where the problem is located
  - the DL-Prover “Hermit” (DL only, no SPARQL) provides better functionality for ontology management.

11.1.1 The Rule-Based OWL Reasoner in Jena

- Jena's built-in `OntModelSpec`s activate the internal rule-based engine with corresponding specifications
- the most powerful one is OWL_DL_MEM_RULE_INF
  `https://jena.apache.org/documentation/inference/#owl`
- Extends the RDFS reasoner,
  "since RDFS is not a subset of the OWL/Lite or OWL/DL languages the Jena implementation is an incomplete implementation of OWL/full"
  ... means that OWL Full is an upper bound of it, but it does eben not (really) *cover* OWL-DL, nor OWL-Lite, nor OWL-RL.
- call
  `jena -inf -if` *`inputfile`* `-qf` *`queryfile`*
- also available in the Web interface

The Rule-Based OWL Reasoner

- Strategy: an instance-based reasoner
- Reasoning about class hierarchy: prove things like "$C_1 \sqsubset C_2$":

  add a "prototypical individual" x_{C_1} (which does not have any other properties) to C_1, apply all rules (model completion), and check whether $C_2(x_{C_1})$ is concluded.
  (These individuals should not longer be visible in user queries since Jena 2.1)
- "The OWL_Mini reasoner ... omits the forward entailments from minCardinality/ someValuesFrom restrictions - that is it avoids introducing bNodes which avoids some infinite expansions"

  ... that tells how to trap the "full" OWL_RULE reasoner with "parents"
  (cf. Slide 620)
  - Tries to provide more than OWL-RL (which excludes SomeValuesFrom specifications because they introduce implicit objects, which requires a blocking algorithm)
  - even OWL Lite includes SomeValuesFrom, even with functionality restriction ($\mathcal{SHIF}$).

The Rule-Based OWL Reasoner (cont'd)

- OWL_Micro reasoner: "RDFS plus the various property axioms, intersectionOf, unionOf (partial) and hasValue. It omits the cardinality restrictions and equality axioms, which enables it to achieve much higher performance."
- it seems that the OWL_RULE reasoner also ignores cardinalities, including "minCardinality 1" (example: (parents, 2), and also (parents, 1)).
- "The critical constructs which go beyond OWL/lite and are not supported in the Jena OWL reasoner are complementOf and oneOf. As noted above the support for unionOf is partial (due to limitations of the rule based approach) but is useful for traversing class hierarchies."
  $\Rightarrow$ No negation, very restricted disjunction (not as choice, only as union), no case-based reasoning

  (cf. the parricides, the meals-wine-ontology from Deductive Databases [Exercise], and win-move)
- "Even within these constructs rule based implementations are limited in the extent to which they can handle equality reasoning - propositions provable by reasoning over concrete and introduced instances are covered but reasoning by cases is not supported."

The Rule-Based OWL Reasoner (cont'd)

Note: Datalog with well-founded semantics, which is polynomial, solves win-move (but not the others).

- the OWL_RULE-Reasoner does not support cardinalities that are necessary to close the "edge" relation.
- even with this, testing membership in AllValuesFrom(edge, LoseNode) would not be possible/easy: [if find-n-edges-to-"lose" then "win"] could be encoded in a set of rules, but the well-founded evaluation is hard.

SOMEVALUESFROM – TRAPPING THE OWL REASONER

Recall the example for the blocking strategy (Slide 519): every person has two parents, which are again persons.

- `jena -q -pellet -qf infinite-parents.sparql` ... pellet correctly does the blocking,
- `jena -q -inf -qf infinite-parents.sparql` ... ... runs forever.
- Aside: when deleting the last 2 lines from `infinite-parents.sparql` ($\mathsf{HasParent} \equiv \exists \mathsf{hasParent}.\top$), the computation finishes with the same output as pellet (and some blanknodes).
  The still existing $\mathsf{Person} \sqsubseteq \exists^{=2}\mathsf{hasParent}.\top$ is ignored. Replacing this by $\mathsf{Person} \sqsubseteq \exists^{=1}\mathsf{hasParent}.\top$ (cardinality 1) leads to some more blank nodes, but $\mathsf{Person} \sqsubseteq \exists \mathsf{hasParent}.\mathsf{Person}$ (someValuesFrom Person) again fails.

The OWL_Mini Reasoner: ignores SomeValuesFrom

- With the OWL_Mini Reasoner, the example runs, but the SomeValuesFrom is completely ignored: Nobody is a HasParent (because then, its definition is empty).
  Java code see next slide.

```
import org.apache.jena.ontology.OntModelSpec;
import org.apache.jena.query.Query;
import org.apache.jena.query.QueryExecution;
import org.apache.jena.query.QueryExecutionFactory;
import org.apache.jena.query.QueryFactory;
import org.apache.jena.query.ResultSet;
import org.apache.jena.query.ResultSetFormatter;
import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.ModelFactory;
import org.apache.jena.util.FileManager;

public class JenaIntRuleMicro {
  static String filepath = "/home/may/teaching/SemWeb/RDF/";
  public static void main(String[] args){
    Model m = FileManager.get().loadModel(filepath + "infinite-parents.n3");
    OntModel ontmodel = ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM_RULE_INF, m);
    // OntModel ontmodel =
    //   ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM_MICRO_RULE_INF, m);
    ontmodel.prepare();
    System.out.println(" ... prepared the model ...");

    String q = "prefix : <foo://bla#> " +
        "select ?A ?C ?X " +
        "where {{?A a :Parent} UNION {?C a :HasParent} UNION {:kate :parent ?X}}";

    Query qu = QueryFactory.create(q);
    QueryExecution qe = QueryExecutionFactory.create(qu, ontmodel);
    ResultSet results = qe.execSelect();
    ResultSetFormatter.out(results);
} }
```

[Filename: Java/JenaIntRuleMicro.java]

Code/Test Example: Deduction tracing in the Jena Rule-Based OWL Reasoner

- next slide: Java test example for tracing deductions (transitivity: descendants)
- getDeductionsModel() contains lots of trivial OWL axioms.
- useful conclusions (and lots of other conclusions) are in diffModel = ont.difference(rawModel).difference(dedModel);
- no derivation information available.

Further evaluation of its functionality: Exercise.

```
  static String filepath = "/home/may/teaching/SemWeb/RDF/";
  public static void main(String[] args){
    Model m = FileManager.get().loadModel(filepath + "descendants.n3");
    OntModel ontmodel = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM_RULE_INF, m);
    Reasoner reasoner = ontmodel.getReasoner();
    reasoner.setDerivationLogging(true);

    Model dedModel = ontmodel.getDeductionsModel();
    if (dedModel == null) System.out.println("DedModel is null");  // it is null ...
        else dedModel.write(System.out,"N3");   // viele triviale OWL-Axiome im DedModel
    Model rawModel = ontmodel.getRawModel();
    Model diffModel = ontmodel.difference(rawModel).difference(dedModel);
    System.out.println("---- DiffModel: ----------------------");
    diffModel.write(System.out,"N3");  // vieles, incl die sinnvollen Ergebnisse im DiffModel

    PrintWriter out = new PrintWriter(System.out);
    for (StmtIterator i = diffModel.listStatements(null,
        ontmodel.getProperty("foo://bla/meta#descendant"),
        (RDFNode) null); i.hasNext(); ) {
        Statement s = i.nextStatement();
        out.write("Statement is " + s + "\n");
        Iterator <Derivation> ds = ontmodel.getDerivation(s);
        if (ds == null) System.out.println("DerivationsIterator is null");
        else for (Iterator<Derivation> i2 = ontmodel.getDerivation(s); i2.hasNext(); ) {
            Derivation deriv = (Derivation) i2.next();
            deriv.printTrace(out, true);   // derivations are not null, but empty
    }   }
    out.flush();
}}
```

[Filename: Java/JenaIntRuleDeductions.java]

11.1.2 The Generic Rule-Based Reasoner in Jena

- allows three types of rule handling (cf. Lecture “Deductive Databases”):
  - forwards-chaining bottom-up (T_P^ω-style; efficient well-known RETE algorithm)
    - if a *ground instance of the rule body matches facts in the DB*, the instance of the head atom is derived to extend the RDF graph with additional derived facts.
    - with *hybrid rules* (cf. Slide 634; rules whose head is a (backward) rule): derive additional rules that extend the program
  - backwards-chaining top-down (Prolog-SLG resolution, basically like XSB including tabling etc.)
    - if the *rule head matches a query*, it is tried to find answers for the body (whose atoms may be basic predicates or match the heads of other rules)
  - hybrid: backward-and-forward mixed. The user can for each rule define how it should be interpreted. (default configuration)

Call with semweb.jar - command line arguments

- -if, -q, -qf as usual
- -rf *rulefile*
- -fw, -bw, -bwfw or -hybrid

Basic Rule Syntax: File or String with ...

- `https://jena.apache.org/documentation/inference/#RULEsyntax`
- rules as "[*head* <- *body*]" or "[*body* -> *head*]" ,
  or "*head* <- *body* ." or "*body* -> *head* ." (as in Datalog),
  - -fw reasoner interprets only "->" rules, multiple head atoms are allowed (left-to-right exec),
  - -bw reasoner interprets all rules (handle then as backward rules, Prolog-style),
  - -hybrid interprets both types in 2 stages: fw first, then afterwards bw
- triple patterns as "(s p o)" with variables ?x as in SPARQL; equal(?x,?y)
- "?x a ?c" is not understood, use "?x rdf:type ?c" instead
  (note: using "a" does not cause an error message, but simply nothing is matched)
- body: lessThan(?x,?y), sum(?a,?b,?c) (if safe, cf. Deductive Databases lecture), now(),
- only in forward rules:
  - body: constructors like strConcat(...,?res), uriConcat(...,?res), makeSkolem(...)
  - updates like remove(n) in the head (removes ground instance of the n-th body term),
- handling/iterating over RDF lists,
- no aggregation (?)

Rules Example

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
[ (?x rdf:type :Bigcountry)
   <-
  (?x rdf:type :Country), (?x :population ?p), greaterThan(?p, 10000000) ]
```

[Filename: RDF/rule-bigcountries-bw.rl]

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>.
[ (?x rdf:type :Country), (?x :population ?p), greaterThan(?p, 10000000)
   ->
  (?x rdf:type :Bigcountry) ]
```

[Filename: RDF/rule-bigcountries-fw.rl]

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?C ?N
# from <file:mondial-europe.n3>
where { ?C a :Country .
        optional {?C a :Bigcountry; :name ?N}}
```

[Filename: RDF/rule-bigcountries.sparql]

```
jena -if mondial-europe.n3 -rf rule-bigcountries-bw.rl \
     -qf rule-bigcountries.sparql -bw
```

Rules Example

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>
[ (?x rdf:type :Country), (?x :population ?p), greaterThan(?p, 10000000),
  strConcat('A','B',?res)    ### to test strConcat in fwd/bwd eval
    -> (?x rdf:type :Bigcountry), (?x :testprop ?res) ]
[ (?x :bigneighborwith ?y)
  <- (?x rdf:type :Bigcountry), (?x :neighbor ?y), (?y rdf:type :Bigcountry) ]
```

[Filename: RDF/rule-neighbor-bigcountries.rl]

```
prefix : <http://www.semwebtech.org/mondial/10/meta#>
select ?N ?X ?TT
# from <file:mondial-europe.n3>
where {{ <http://www.semwebtech.org/mondial/10/countries/D/>
         :bigneighborwith ?X }
       union { ?C a :Bigcountry; :name ?N OPTIONAL { ?C :testprop ?TT}}
      }
```

[Filename: RDF/rule-neighbor-bigcountries.sparql]

- -bwfw/hybrid fill both (also when union line commented out) and sets :testprop
- -fw fills only “Bigcountry” (?N) (i.e., ignores backward rules)
- -bw fills both, but ignores the strConcat literal in the fwd rule, does not fill :textprop

Rules Example

- same rule, other directions:

```
@prefix : <http://www.semwebtech.org/mondial/10/meta#>
[ (?x rdf:type :Bigcountry)
  <- (?x rdf:type :Country) , (?x :population ?p), greaterThan(?p, 10000000) ]

[ (?x rdf:type :Bigcountry), (?x :neighbor ?y), (?y rdf:type :Bigcountry)
  -> (?x :bigneighborwith ?y)  ]
```

[Filename: RDF/rule-neighbor-bigcountries2.rl]

- -bw fills both (i.e. also evaluates forward rules, then in backward direction)
- -bwfw fills only “Bigcountry” (?N)
- -fw fills nothing, (ignores backward rules, and the “lower” rule is backward)

Reasoners' Behavior

- Leveling of the hybrid reasoner:
  - forward rules first (filling "views")
  - backwards rules afterwards
- Backward-chaining reasoner also evaluates forward rules, then in backward direction,
- Forward-chaining reasoner ignores backward rules.

Including other rule sets

- `@include` <*otherrulefile*> in the rule file.
- instead of <*otherrulefile*>, also keywords RDFS, OWL, OWLMini, and OWLMicro are allowed to preload the respective rule sets.

Loading Rules in Java

- List<Rule> rules = Rule.rulesFromURL(*http:-URL or file*);
- String ruleSrc = "*list of rules as string*"
  List rules = Rule.parseRules(rulesSrc);

Tabling

- Analogously to Prolog/Datalog: In top-down/backward evaluation, intermediate results are cached ("tabled").
  This is especially necessary for stratified negation, but also more efficient whenever some subgoal can be reused.
- tableAll(), table(P). If any property is tabled, goals such as (?A, ?P, ?X) will all be tabled because the property variable might match one of the tabled properties.
- Syntax in rule file: `[ -> tableAll() ]` or `[ -> table(:bigneighborwith) ]`
- The tabled results of each query are kept indefinitely. Queries can exploit all of the results of the subgoals involved in previous queries. In essence we build up a closure of the data set in response to successive queries.
- tabling: reset()
- When the inference model is updated by adding or removing statements all tabled results are discarded by an internal reset() and the next query will rebuild the tabled results from scratch.
- Java Code next slides. Tabling of backward reasoning is not stored in the DeductionModel.

```
import org.apache.jena.rdf.model.InfModel;
import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.ModelFactory;
import org.apache.jena.reasoner.rulesys.GenericRuleReasoner;
import org.apache.jena.reasoner.rulesys.Rule;
import org.apache.jena.util.FileManager;

public class JenaRules {
  static String filepath = "/home/may/teaching/SemWeb/RDF/";
  public static void main(String[] args){
    // Model model = ModelFactory.createDefaultModel();
    Model m = FileManager.get().loadModel(filepath + "mondial-europe.n3");
    List<Rule> rules = Rule.rulesFromURL(filepath + "rule-neighbor-bigcountries.rl");
    GenericRuleReasoner reasoner = new GenericRuleReasoner(rules);
    reasoner.setMode(GenericRuleReasoner.HYBRID);
    InfModel model = ModelFactory.createInfModel(reasoner, m);
    String q = "prefix mon: <http://www.semwebtech.org/mondial/10/meta#>" +
             "select ?N ?X" +
               " where {{ ?Z" + // <http://www.semwebtech.org/mondial/10/countries/D/>" +
             "          mon:bigneighborwith ?X }" +
             " union { ?C a mon:Bigcountry; mon:name ?N }}";
    Query qu = QueryFactory.create(q);
    QueryExecution qe = QueryExecutionFactory.create(qu, model);
    ResultSet results = qe.execSelect();
    ResultSetFormatter.out(System.out, results, qu);

    model.getDeductionsModel().write(System.out,"N3");
  }
}
```

[Filename: Java/JenaRules.java]

```
GenericRuleReasoner reasoner = new GenericRuleReasoner(rules);
reasoner.setMode(GenericRuleReasoner.HYBRID);
reasoner.setDerivationLogging(true);
InfModel model = ModelFactory.createInfModel(reasoner, m);
String q = "prefix mon: <http://www.semwebtech.org/mondial/10/meta#>" +
         "select ?N ?X" +
           " where {{ <http://www.semwebtech.org/mondial/10/countries/D/>" +
         "             mon:bigneighborwith ?X }" +
         "  union { ?C a mon:Bigcountry; mon:name ?N }}";
Query qu = QueryFactory.create(q);
QueryExecution qe = QueryExecutionFactory.create(qu, model);
ResultSet results = qe.execSelect();
ResultSetFormatter.out(System.out, results, qu);

Model dedModel = model.getDeductionsModel();
Model rawModel = model.getRawModel();
Model diffModel = model.difference(rawModel).difference(dedModel); // = bwd derivations
System.out.println("---- Derivations (in the DedModel) : ----------------------");
dedModel.write(System.out,"N3");
System.out.println("---- Derivations (in the DiffModel) : ----------------------");
PrintWriter out = new PrintWriter(System.out);
for (StmtIterator i = diffModel.listStatements(null,
      model.getProperty("http://www.semwebtech.org/mondial/10/meta#bigneighborwith"),
      (RDFNode) null); i.hasNext(); ) {
    Statement s = i.nextStatement();
    out.write("Statement is " + s + "\n");
    for (Iterator<Derivation> i2 = model.getDerivation(s); i2.hasNext(); ) {
        Derivation deriv = (Derivation) i2.next();
        deriv.printTrace(out, true);
    } }
out.flush(); } }
```

[Filename: Java/JenaRulesDedModel.java]

Insights into derivations with the GenericRuleReasoner (Hybrid)

- `Model getDeductionsModel()`:
  - even if no query is stated, the *full deductive closure inc. backward chaining rules is evaluated* when executing such model operations.
  - getDeductionsModel() contains only for forward rule firings. This allows the forward rules to be used separately as if they were rewrite transformation rules (create *new* graph *without* old).
  - Facts derived by the backward-chaining rules: intermediate results are tabled (internally), and added to the InfModel, but neither in the rawModel nor in the deductionsModel.
  - For all derived facts, derivation trees (their T_P^ω-derivation) are available.

Forward rules fire on ontology updates

- With the forward chaining reasoner, if the InfModel is changed (add or remove triples) through the API, this triggers rule evaluations incrementally (RETE algorithm).

HYBRID RULES:
GENERATION OF NEW RULES BY FORWARD RULES GENERATING RULES

- Forward rules are allowed to create new rules (only backward rules) in their *heads*. (Then, use the "[...]" syntax for rules to allow for nesting)
- mostly used to break down second-order patterns to first order instantiations (like e.g., the transitivity pattern):

Example 1

Consider the case of rdfs:subPropertyOf assertions:

```
[   (?a ?q ?b) <- (?p rdfs:subPropertyOf ?q), (?a ?p ?b) .]
```

as a backward rule. For *every* (sub)goal of the form (?x anyprop ?v), the head will match, and the subgoal is replaced by (?p rdfs:subPropertyOf anyprop), (?x ?p ?v), usually only for finding out that *anyprop* does not have subproperties.

Thus, the application cases can be restricted as follows: Add a *hybrid rule* whose outer, forward rule, "fires" for each (p rdfs:subProperty q) fact and creates a *partial instance* of the above rule:

```
[ (?p rdfs:subPropertyOf ?q), notEqual(?p,?q) -> [ (?a ?q ?b) <- (?a ?p ?b) ] ]
```

Thus, for, e.g., a statement (:cityIn rdfs:subPropertyOf :locatedIn) it will add the rule

```
[  (?a :locatedIn ?b) <- (?a :cityIn ?b) ]
```

which matches only subgoals of the form (?x :locatedIn ?c).

Example 2: Transitivity

Transitivity is a typical "syntactically second-order propery" which can be mapped to its first-order instantiated pattern. – Exercise.

Example 3: Property Chains

The “Property Chain/Role Chain” pattern is another example for such a hybrid rule:

- the owl:PropertyChain construct (cf. Slide 502) is internally (and in RDF/XML, see Slide 503) mapped to an RDF list, but this is immediately “consumed” by the parser and appropriate knowledge is added to the model.

```
[ owl:propertyChain (:brotherOf :hasChild) ]
  rdfs:subPropertyOf :uncleOf.
```

- For an independent rule-based handling, use another suitable presentation by RDF triples that connect the chain property with its constituents.
  Note that these triples connect properties, so they are not common RDF properties, but must be declared as owl:AnnotationProperties, which are ignored by OWL reasoning, but nevertheless accessible for graph-level queries with SPARQL and in rule bodies:

```
:propchainFirst a owl:AnnotationProperty.
:propchainSecond a owl:AnnotationProperty.
:hasUncle :propchainFirst :hasParent ; :propchainSecond :hasSibling .
```

- The forward rule pattern must then match the above and generate appropriately instantiated backward rules, here

```
[ (?X :uncleOf ?Z) <- (?X :brotherOf ?Y, ?Y :hasChild ?Z) ]
```

Example: Property Chains (cont'd)

```
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <foo://bla#>.
  :john :hasChild :bob; :hasSibling :paul, [].
  :hasSibling a owl:SymmetricProperty.
  :paul a [a owl:Restriction; owl:onProperty :hasChild; owl:minCardinality 1].
  :hasChild owl:inverseOf :hasParent.
:propchainFirst a owl:AnnotationProperty.
:propchainSecond a owl:AnnotationProperty.
:hasUncle :propchainFirst :hasParent ; :propchainSecond :hasSibling .
```

[Filename: RDF/uncle-rule-rl.n3]

```
@prefix : <foo://bla#>.
@include <OWL>.
[ (?r :propchainFirst ?p), (?r :propchainSecond ?q) ->
         [ (?x ?r ?y) <- (?x ?p ?z) (?z ?q ?y)] ]
```

[Filename: RDF/uncle-rule-rl.rl]

```
prefix : <foo://bla#>
select ?P ?U
where { ?P :hasUncle ?U }
```

[Filename: RDF/uncle-rule-rl.sparql]

call: jena -hybrid -if uncle-rule-rl.n3 -rf uncle-rule-rl.rl -qf uncle-rule-rl.sparql

Functors in Rules [additional syntax, ignore]

Functor syntax within rules does allow creation of nested "data structures": $f(a, b, c, d)$

- Datalog: allows arbitrary n-ary predicate symbols in rule heads (EDB)
- RDF: allows only triples,
  complex relationships cannot be described in a single atom,
  $\rightarrow$ requires several triples for reification
- Functors: *auxiliary syntactical sugar* allowed to create in heads of *forward rules* and in rule bodies.

$\Rightarrow$ such terms are internal to the rule evaluation,
finally there must be a rule that creates triple instances into the graph.

- functor terms may be nested.

Generic Rule Reasoner + OWL/RDFS rules

`https://jena.apache.org/documentation/inference/#RDFSPlusRules`

- use the GenericRuleReasoner to combine rules for RDFS or OWL and user-defined rules:

```
reasoner.setOWLTranslation(true);
reasoner.setTransitiveClosureCaching(true);
```

- default Jena RDFS and OWL rulesets use the Hybrid rule engine. The hybrid engine is itself layered, forward rules do not see the results of any backward rules.
- all inferences that must be seen *by the RDF/OWL rules* must be forward, all the inferences which need the results of the RDFS/OWL rules must be backward.
  - "complete" RDF graph by forward-chaining rules (like uncleOf etc.)
  - then fwd-bwd predefined OWL rules,
  - further backward-chaining rules building upon OWL conclusions.

Conclusion

- Rules cannot have negative literals in the body

$\Rightarrow$ the rules implementing the OWL fragment can also not use negation
(OWL has open world, rules have closed world)

- ... not more than positive Datalog restricted to triples??

Chapter 12
Conclusion and Outlook

What should have been learnt:

- Formal Logic: interpretations, model theory, first-order logic
- Deductive systems: Datalog, minimal model semantics
- reasoning: tableau calculi
- RDF as a special, simple data model; URIs
  representations: Turtle and RDF/XML
- DL as another logic, Open World
- “database” vs. “knowledge base”
- OWL as “DL alive”

SEMANTIC WEB DATA: XML; RDF AND OWL

In contrast to XPath/XQuery, XSLT, XML Schema, XLink etc., RDF and OWL are *not* languages *"inside"* the XML world, but are concepts of their own that have - incidentally- also an XML syntax.

The combination of XML data and RDF/RDFS/OWL concepts is the base for the *Semantic Web*.

A Semantic Web application e.g. exists of

- a "central" portal that uses the following things:
- a set of ontological (OWL, RDFS) sources,
- a set of RDF sources,
- reasoning (using OWL and RDFS information),
- a semantical description of itself for allowing others to use it.