SemTech 2011 Tutorial

SPARQL and Inferencing
Overview

- Introduction and Overview
- Using SPARQL and SPARQL CONSTRUCT for reasoning
- SPARQL-BasedReasoning technologies
- Data constraint validation
- Merging Reasoning and OWL inference
- Reasoning with insert/delete/update
- Summary and Conclusions
Goals and Outcomes of Tutorial

- Broaden perspective on what “inference” means
  - much, much more than OWL
- Increase understanding of SPARQL (and SPARQL 1.1)
  - query language for data, including updating data
- Improve understanding of OWL inferences
  - understand the underlying mechanics of OWL through SPARQL
- Understanding the continuum of inference and rules
  - systems of inference (OWL, SPIN)
- Data integrity constraints
  - consistency checking vs. data constraints
Inferencing in the Semantic Web

Inference = New Triples from Old

- **RDFS and OWL inferencing**
  - standard inferences; subproperty, subclass, etc.

- **Business Rules**
  - Customers who fly more than 25K miles per year get Silver Status

- **Computations**
  - Duration = return date minus departure date

- **Data Quality**
  - Departure date must precede return date
SPARQL

- What’s expected: a query language
  - SELECT variables from matches

- What’s expected: a data update language
  - SPARQL 1.1 Update

- What’s not expected: built-in inference mechanism
  - CONSTRUCT

- What’s not expected: general-purpose expression language
  - FILTER, BIND, aggregates, project, etc.

- What’s missing: functional decomposition
  - SPIN functions

- What’s missing: stored procedures
  - SPARQLMotion scripts

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way

This is the main topic of the tutorial – the other topics will be explored along the way
SPARQL CONSTRUCT

- SPARQL SELECT returns a set of variable bindings

```
SELECT ?city ?population
WHERE
{ ?city a g:City .
  ?city skos:broader* g:United_States .
  ?city g:population ?population
} ORDER BY ?population
```

- SPARQL Construct returns triples...
  - returns triples – these can easily be placed in asserted or inferred graph

```
CONSTRUCT {?city a g:LargeCity}
WHERE
{ ?city a g:City .
  ?city skos:broader* g:United_States .
  ?city g:population ?population
  FILTER (?population > 100000)
}
```

Note that resources and variables can be mixed in CONSTRUCT

- `g:New_York_City rdf:type g:LargeCity`.
- `g:Los_Angeles  rdf:type g:LargeCity`.
- `g:Chicago    rdf:type g:LargeCity`.
Reasoning with RDF

- Inference with RDF data
  - add triples
  - if someone went to a college and that college is in a state
    - infer that person lived in that state

- RDFS Reasoning
  - given a set of data
    - ...add some triples
  - Open World Assumption inference rule for rdfs:domain

CONSTRUCT {?person :livedIn ?state}
WHERE
  ?college :inState ?state
}

CONSTRUCT {?s rdf:type ?cls}
WHERE
{  ?pred rdfs:domain ?cls
  ?s ?pred ?o .
}
OWL Reasoning

- given a set of data
  - ...add some triples
- OWL can be decomposed into a set of rules
  - aside from a few exceptions that are beyond the scope of a short tutorial
- e.g. inverse property rule

```
CONSTRUCT {?inv ?pred ?s}
WHERE
{ ?s ?pred ?o .
}
```
OWL never intended as a general-purpose reasoning engine

- must augment with rules for any mathematical computation, etc.

<table>
<thead>
<tr>
<th>Jena Rule</th>
<th>SPARQL</th>
</tr>
</thead>
</table>
When, Where, and How Inferences are Applied

- Rule execution (when)
  - System runs rules as a set, usually forward-chaining
  - Run OWL inferences then rules, or vice-versa
  - SPIN defines rule sets
  - SPARQLMotion defines execution pipelines for rules

- Rule scoping (where)
  - Jena rules, OWL, RIF (formerly SWRL) have global scope
    - E.g. the entire RDF graph
  - SPIN uses a class scoping
  - SPARQLMotion can apply to arbitrary sets of triple

- Rule engine (how)
  - Engine applies rules
  - Results applied to inferred graph
  - SPIN & SPARQLMotion can infer or assert
OWL 2 RL profile

- designed for rule-based systems, defined in a triple-based syntax
- directly implementable in SPARQL

transitive subproperty rule

```
# prp-trp
CONSTRUCT {
}
WHERE {
  ?p a owl:TransitiveProperty .
```
Consistency Checking vs. Data Constraints

- Consistency checking detects model inconsistencies
  - must find a way to create an inconsistency
  - e.g. checking for errors in a U.S. SSN is difficult
    - can define “hasSSN some xsd:string[pattern "[0-9]{3}-[0-9]{2}-[0-9]{4}" ]”
    - ...but how to detect an inconsistency when the string is not of that pattern?
    - e.g. define the model to classify members not meeting the pattern as a member of owl:Nothing or members of disjoint classes, etc.

- Data constraints
  - any data pattern can queried for consistency checks

```sparql
ASK
WHERE
{   ?rsc :hasSSN ?ssn .
    FILTER regex(?ssn, "[0-9]{3}-[0-9]{2}-[0-9]{4}" )
}
```
Reasoning and Data Updates

- Monotonic reasoning
  - can add new triples ...but cannot delete any
  - modification and deletion of data not in scope for OWL/RIF/Jena Rules/etc.
    - SPARQLMotion can use SPARQL manipulate graphs to modify data
    - SPARQL 1.1 Update

- Broadening the Semantic Web notion of inference
  - e.g.: “last edited date”
  - modification as a delete/insert combination

```sparql
WITH <http://topquadrant.com/ns/examples/demo>
DELETE
{ ?s ex:lastUpdated ?lasttime . }
INSERT
{ ?s ex:lastUpdated ?time . }
WHERE
{ ?s ex:lastUpdated ?lasttime . }
```
SPARQL RECAP
SPARQL

- SPARQL as a query language
  - SPARQL is to RDF graphs what SQL is to relational models
  - underlying model is an RDF graph – i.e. triples
    - ...not relational tables
  - e.g. no notion of “join” because there are no tables to join
    - instead use triple patterns
    - match patterns to data

- Not just queries
  - various query language manipulations
  - CONSTRUCT, INSERT
  - functions for manipulating data
    - string manipulation, data types, logical and math operations, etc.
SPARQL in Brief

- SPARQL is a query language for graph models
  - ...just as SQL is a query language for relational models

subset of graph representing John Kennedy Jr. and Maria Shriver’s uncles
SPARQL is a query language for graph models

- Who was Caroline Kennedy father?
- Who are Maria Shriver’s uncles?
- Who is a mother of whom?
Who was Caroline Kennedy father?

SELECT ?x
WHERE
{ :CarolineKennedy :parent ?x .
}

?x
John Kennedy
Jacqueline Bouvier
SPARQL Example

Who was Caroline Kennedy father?

SELECT ?father
WHERE
{ :CarolineKennedy :parent ?father .
}

We have specified a graph pattern to match

:CarolineKennedy :parent ?father :gender :male

?father

John Kennedy
Example queries

Find grandparents of John Kennedy Jr

```
SELECT ?gparent
WHERE {
}
```
SPARQL and Graph Matching

- SPARQL specifies a graph (in triples) to match

```sparql
SELECT ?gparent
WHERE {
}
```
ASK returns a Boolean value
- based on whether WHERE clause finds data

Were any plays written after 1600?

Most useful in programs
- SPIN
- SPARQLMotion, JSP, etc.

Result displayed in status bar
Query Forms

- **SELECT**
  - returns matches for specified variables

- **CONSTRUCT**
  - returns a graph that includes triples constructed from specified variables

- **ASK**
  - true if a match to the graph pattern is found

- **DESCRIBE**
  - returns a graph (instead of selecting “columns”)}
Thus far, have used SPARQL to:
- select a set of variables in a result set
- return a Boolean value

Note that SELECT returns a result set (basically a table)
- can return a triple instead
- ...but need to define the triple

```sparql
CONSTRUCT {?person a :Mother} 
WHERE 
}
```

classify a female person with a child as a member of :Mother
CONSTRUCT

- Building new sets of triples based on patterns
  - ...and asserting those triples

```
CONSTRUCT {?person a :Mother}
WHERE
}
```

Any woman with children is a Mother

Returns an RDF graph
...but ontology (file) is not changed
CONSTRUCT

- Building new sets of triples based on patterns
  - ...and asserting those triples

```
CONSTRUCT { ?person rdf:type :Mother }
WHERE
}
```

- Choose desired triples
- Assert into ontology
A “sister” is a female sibling

- that person is also “sister of” all her siblings
- i.e. an inverse relationship

# from sister's perspective

CONSTRUCT

}

WHERE

  FILTER (?person != ?sibling) .
}
Follow-Along Exercise

- **Using CONSTRUCT**
  - import file on universities
    1. construct triples to give resource a type
    2. construct triples to show what states people lived in

1. **CONSTRUCT**
   ```
   { ?state a world:State .
     ?univ a kennedys:College .
   }
   WHERE
   { ?univ world:state ?state .
   }
   ```

2. **CONSTRUCT**
   ```
   { ?person kennedys:livedIn ?state .}
   WHERE
   { ?person a kennedys:Person .
   FILTER (regex (?cl, ?dbul)) .
   }```
INFERENCE IN THE SEMANTIC WEB
Q: What does it mean?

Answer: These inferences can be made

e.g., “if City subClassOf VacationDestination, and NewYorkCity rdf:type City,

...then NewYorkCity rdf:type VacationDestination.”
Members of subclass are also (inferred) members of any super classes.

This reflects ordinary understanding of subclass as set inclusion.

- NewYorkCity
- Newark
- Oxford
- London
- Mt. Rushmore
- Vacation Destination
- City
Domains

- Individuals must belong to domains of any properties used to describe them
  - “hasAccomodation rdfs:domain VacationDestination” means “Only VacationDestinations can haveAccomodation”
- Inference will enforce this by adding rdf:type triples
- Simple rule – two triples match, one triple inferred
- Domain can be undefined (no inferences will be made)
Interpretation as sets

- Remember that classes can overlap arbitrarily!

Diagram:
- VacationDestination
  - Newark
  - NewYorkCity
- Park
  - Arcadia
  - Yellowstone
  - hasAccommodation
Ranges

- Individuals must belong to range of any property that points to them
  - “Only Hotels can be a ‘target’ of hasAccomodation property”
- Inference will enforce this by adding rdf:type triples
- Simple rule – two triples match, one triple inferred
- Ranges can be undefined (no inference will be made)
SubProperty Relationships

- subPropertyOf relates properties to one another
  - Find me a hotel located near Yellowstone. Should that include the hotels locatedIn Yellowstone?

Syntax

:locatedIn a rdf:Property ;
rdfs:subPropertyOf :locatedNear
If YellowstoneLodge locatedIn Yellowstone, and locatedIn rdfs:subPropertyOf locatedNear, YellowstoneLodge locatedNear Yellowstone.
Motivation for Inferences

- RDFS and OWL inferencing
  - standard inferences; subproperty, subclass, etc.
- Business Rules
  - Customers who fly more than 25K miles per year get Silver Status
- Computations
  - Duration = return date minus departure date
- Data Quality
  - Departure date must precede return date
Inferences and RDF

- Using SPARQL CONSTRUCT to create graphs
  - only returned, not applied to any existing model
  - therefore need an additional steps to turn individual SPARQL queries into a rule/inference system

- The process of applying rules for inference
  - when to apply the rules
    - SPARQLMotion – part of the data processing pipeline
    - SPIN – apply rule to all members of the class the rule is defined in
  - what happens to inferred triples
    - inferred graph of model,
    - apply to a file, a data store, etc
RDFS and OWL in SPARQL
Inferences and Rules

- What is an “inference”? 
  - process of reasoning whereby one statement (the conclusion) is derived from one or more other statements (the premises)

- Sounds like a rule...
  - RDFS reasoning is a set of rules applied together

- RDFS can be implemented as rules

```sparql
CONSTRUCT {
  ?rsc a ?supc .
}
WHERE {
  ?rsc a ?cls .
  ?cls rdfs:subClassOf* ?supc .
}
```

- set of facts to infer (note that CONSTRUCT does no inference – need another step to infer these triples)
- premise – query the RDF graph
- subClassOf executed as a transitive property in SPARQL
The subPropertyOf inference

Create new triples using the (super)property. 
I.e. whenever there is a:

```
?subject ?p1 ?object
```
add a:

```
?subject ?p2 ?object
```

when a subPropertyOf definition is found...

find all places where the subproperty is used
- subPropertyOf, subClassOf rules from RDFS
- General rules – applicable to all resources
OWL Profiles

- OWL 2 DL: All Description Logic Constructs available
- **OWL 2 RL**: OWL for Rule Languages
- OWL 2 EL: Large-scale class inferences
- OWL 2 QL: OWL for Query Languages (SQL)

OWL RL can be expressed in Rule languages (like SPARQL!)
hasValue restrictions – inferences1

Disney’s Euro Adventure

owl:Restriction

rdf:type

owl:onProperty

:hasDestination

owl:hasValue

:Rome

:hasDestination

Paris

London

Amsterdam

Madrid

Athens
OWL rules in SPARQL

- “if ?u has the value ?y on property ?p, then ?u is a member of the restriction on property ?p hasValue ?y.”

- OWL restriction represented as Rule in SPARQL

- Also general purpose, applies to any resource
Why SPARQL?

- SPARQL CONSTRUCT can form the basis of rules
  - no need for additional syntax
  - use the query language to specify how to add new facts
  - new facts are described in the same data model (RDF triples)

- Need additional steps
  - apply the constructed triples
    - CONSTRUCT returns a set of triples
  - means to iterate over a set of rules
BUSINESS RULES in SPARQL
“Customers who have flown over 25K miles have Silver Medallion status”

```
CONSTRUCT {
    c a FrequentFlier:Customer .
    c a delta:Silver .
}
WHERE {
    c a ?class .
    ?class (rdfs:subClassOf)* FrequentFlier:Customer .
    c FrequentFlier:earnedMiles ?x2 .
    FILTER (?x2 > 25000) .
}
```

- Representing Silver Medallion as a Class – the customer is a member of this Class
- Specific rule – only applies to Customers
- The WHERE clause restricts this with the first two lines
Attach rules to classes

“The 25K=Silver rule applies to Customers only”

Property spin:rule attaches SPARQL rules to classes

```sparql
:Customer spin:rule "CONSTRUCT {
    ?c a FrequentFlier:Customer .
    ?c a delta:Silver .
}
WHERE {
    ?c a ?class .
    ?class (rdfs:subClassOf)* FrequentFlier:Customer .
    ?c FrequentFlier:earnedMiles ?v2 .
    FILTER (?v2 > 25000) .
}"
```
SPIN = SPARQL Inference Notation

- Attach rules to classes
- “The 25K=Silver rule applies to Customers only”
- Property spin:rule attaches SPARQL rules to classes

:Customer spin:rule "CONSTRUCT {
    ?c a FrequentFlier:Customer .
    ?c a delta:Silver .
}
WHERE {
    ?this FrequentFlier:earnedMiles ?v2 .
    FILTER (?v2 > 25000) .
}"
Organizing Rule-based Inference

- How do we process these rules as a set?
  - Business could have more rules – e.g., “Silver Medallion customers are eligible for upgrades”
  - Inference means all of these rules need to be executed

- TopSPIN is an inference engine
  - execute a set of rules
  - repeat until no new triples are created
  - defined in a RDFS/OWL class context
    - rules are defined in a class
    - iterated over all instances
COMPUTATIONS
Computations

- Given a person’s birthdate, how old are they?
- Given a distance in feet, how far in meters?
- Given a departure date and a return date, how long was the trip?

- Computations are dynamic, and depend on other data.
- Define computations in SPARQL
Compute duration

- Compute information based on data
- Assert computation (in a new property)
- Using library functions as needed
Incremental Inferencing

- Make change to inferred values as source values change
- Inferred values change, too
DATA QUALITY
SPIN Constraints

- Attached to classes (like rules)
- Boolean constraints “is this true?”
- Violations are flagged
- E.g., “You can’t return from a trip before you leave”
Dynamic evaluation of Constraints

- If changes violate a constraint, a warning shows up
- Mouse over the warning for an explanation
SPARQL Reasoning Technologies
The term “inference” often assumes RDFS or OWL reasoning
- rules defining subClassOf entailments, someValuesFrom, etc.
- general prescription: given a pattern of data, create these triples
  - ...it’s all about the triples

Rule processing in RDF has the same general form
- SPARQL CONSTRUCT is the basis of a rule language
- We have a way to define the rules, but when are they applied?

IF a person is female and has children
THEN person is a Mother

CONSTRUCT {?person a :Mother}
WHERE
{ ?person a :Person .
 ?person :child ?child }
Reasoners distinguish between the asserted and inferred graphs:
- Asserted: those defined in a model
- Inferred: those defined by inference

Inferred triples are not saved – must re-run inference:
- E.g., editors display union of asserted and inferred graphs.

Inferred data shown with violet background.
When, Where, and How Inferences are Applied

- **Rule execution (when)**
  - system runs rules as a set, usually forward-chaining
  - run OWL inferences then rules, or vice-versa
  - SPIN defines rule sets
  - SPARQLMotion defines execution pipelines for rules

- **Rule scoping (where)**
  - Jena rules, OWL, RIF (formerly SWRL) have global scope
    - e.g. the entire RDF graph
  - SPIN uses a class scoping
  - SPARQLMotion can apply to arbitrary sets of triple

- **Rule engine (how)**
  - Engine applies rules
  - results applied to inferred graph
  - SPIN & SPARQLMotion can infer or assert
RDF-Based Rule Technologies

- **SPIN (W3C member submission)**
  - SPIN Rules – create a SPARQL CONSTRUCT query on a class
    - ...and apply to all members of that class
  - SPIN Constraints – define and apply data constraints on class instances
  - SPIN features for defining SPARQL abstractions
    - functions and templates
  - RDF syntax for SPARQL – very important

- **SPARQLMotion (TopBraid technology)**
  - scripting language for RDF operations
    - define data processing pipelines
  - manipulate sets of triples in memory
    - import & filter arbitrary sets of triples
    - rules on results
    - output to various places
SPIN Example

- SPIN allows for class-specific rules
  - apply to class instances
  - ...including rdfs:subClassOf entailments
  - executed as inference
  - combine with OWL RL profile to run OWL + rules

Applied to kennedys model, these rules apply to :Person class and all subclasses

?this is a reserved variable name: it represents the member the rule will be bound to when executing the query. (only place SPIN departs from SPARQL standard)

all constructed triples added to the inferred graph

SPIN function – SPARQL query that returns the current year

SPIN representation of SPARQL: if a property name changes, the query automatically changes
Define pipeline to process data using SPARQL

- e.g. saving inferences
- run as a script
  - combine flexibly with other inferences

geo:lat and geo:long cause all resources with lat/long values to be members of geo:SpatialThing – Filter these out:

Not all OWL inferences are useful. Filter out subClassOf and sameAs triples

Save union of triples from original model and (filtered) inferred triples
- note this asserts the inferences in this model

Save only the inferences
- (remember replace=true in ApplyTopSPIN)
More on SPIN

- **SPARQL RDF syntax**
  - RDF syntax for SPARQL queries

- **SPARQL Functions**
  - Functions, defined in SPARQL, returning a value

- **SPARQL Templates**
  - Parameterized functions

- **SPARQL Rules**
  - SPARQL Inference: SPARQL-based reasoner (includes OWL 2 RL)
  - SPARQL Constraints: constraint violation warnings
  - SPARQL Constructors: execute rule when creating an instance

- **SPARQL Magic Properties**
  - Property defined as a SPARQL query, return multiple bindings

- **SPARQL JavaScript Extensions**
  - Interface to JavaScript for SPARQL Functions
SPIN Constraints

- Warnings are displayed on applied resources

Two warnings

Either kenedys:spouse or kenedys:gender could be the problem
Constraint Violations

- Use spin:ConstraintViolation to target where warning is placed
  - use CONSTRUCT to build constraint warning triples

```
CONSTRUCT {
  _:b0 a spin:ConstraintViolation .
  _:b0 spin:violationRoot ?this .
  _:b0 spin:violationPath kennedys:spouse .
  _:b0 rdfs:label "Always blame it on the spouse" .
}
WHERE {
  FILTER (?gender = ?spouseGender) .
}
```

gender no longer displays a warning
Define functions using SPARQL queries

- use in FILTER and LET clause
- from previous example, create a getcurrentYear function

CONSTRUCT { ?this :age ?age . }
WHERE {
  ?this kennedys:birthYear ?birthYear .
  LET (?currentYear := :getCurrentYear()) .
  LET (?age := (?currentYear - ?birthYear)) .
}

Body of getcurrentYear

SELECT ?year
WHERE {
  LET (?str := xsd:string(afn:now())) .
  LET (?sub := fn:substring(?str, 0, 4)) .
  LET (?year := xsd:integer(?sub)) .
}

- using afn:now() as before
- returns ?year
- will return first bound
Reusable SPARQL Queries

- SPIN templates define parameterized SPARQL queries
  - used with spin:rule, spin:constraint, etc.
  - a generalization of SPIN Functions
- Let’s look at rules for aunts and uncles

except for gender & property name
these are the same!
Defining SPIN Templates

- Define a subclass of spin:Templates
- Define arguments
  - same as with functions

Note that local name of arguments are the variable names
Want a creation date for each newly created resource

- apply SPIN construct to rdfs:Resource

afn:now() is a built-in function to get current time in xsd:date format
More on SPARQLMotion

- Script defines data processing steps
  - properties define relationships between modules
  - ‘next’ means result triples from one module available to next
    - it’s always RDF triples – sometimes referred to as the “triple stream”
    - ...plus variable bindings, which are available from binding to end of script
  - execution order determined by pipeline’s ‘next’ properties

- Also the infrastructure for Web service definitions
  - script defines what gets loaded into memory for service
    - also passes parameters
  - ...process and export to some format (normally XML or JSON)
  - name with a SPIN function
ApplyConstruct applies a SPARQL CONSTRUCT query to the data input.

- Output of module can be either:
  - Union of input triples and constructed triples (replace=false, the default)
  - Only constructed triples (replace=true)

Arguments accepted by script:
ApplyConstruct

- Fine-grained control over execution of SPARQL rules
  - use to create sets of triples to run rules over
  - use the rules to create the sets of triples
  - ApplyConstruct and FilterByConstruct

- Saving inferred triples
  - easily accomplished within pipeline
  - view rules not just as inferences, but rules that are applied to data
Execution Order of Rules

- Ordered sequences of rules
  - e.g. make sure :age is computed before rules using that value
  - some scenarios are covered by SPIN iterations
- spin:nextRuleProperty defines the next rule property to execute
  - create sub-properties of spin:rule
    - (any prop that encodes SPIN RDF)
  - define an ordering between them

Want to run rule sets 1 and 2 before spin:rule

```
sp:rs1 subPropertyOf sp:rule .
sp:rs2 subPropertyOf sp:rule .
sp:rs1 sp:nextRuleProperty sp:rs2 .
sp:rs2 sp:nextRuleProperty sp:rule .
```
Rule Sets and Max Iterations

- Interaction between spin:nextRuleProperty and spin:rulePropertyMaxIterationCount

<table>
<thead>
<tr>
<th>Want to run rule sets 1 and 2 before spin:rule, only run twice</th>
</tr>
</thead>
<tbody>
<tr>
<td>spin:rs1 subPropertyOf spin:rule .</td>
</tr>
<tr>
<td>spin:rs2 subPropertyOf spin:rule .</td>
</tr>
<tr>
<td>spin:rs1 spin:nextRuleProperty spin:rs2 .</td>
</tr>
<tr>
<td>spin:rs2 spin:nextRuleProperty spin:rule .</td>
</tr>
<tr>
<td>spin:rs1 spin:rulePropertyMaxIterationCount 2 .</td>
</tr>
</tbody>
</table>

Rules executed as:
1. Pass 1:
   a. execute all rules in spin:rs1
   b. execute all rules in spin:rs2
   c. execute all rules in spin:rule
2. Pass 2:
   a. execute all rules in spin:rs1
   b. execute all rules in spin:rs2
   c. execute all rules in spin:rule

If also have:
   spin:rs2 spin:rulePropertyMaxIterationCount 2 .
   ...then 2b would be skipped
Rule Sets in SPARQLMotion

- Rule sets can also be ordered with SPARQLMotion scripts
  - sml:predicate of ApplyTopSPIN can contain any property
  - (subproperty of spin:rule)
  - design script to run the rule sets
    - i.e. multiple ApplyTopSPIN modules with different rule sets specified
  - can accumulate the rule inferences or run on any data set desired
    - ...as specified in triple stream (sm:next links)

- This is more flexible, as one could run a rule set, then check some values to determine whether the other rule set needs to be executed, or which rule set, etc.
Data Constraint Validation
Consistency is checked within a model
- given pre-defined rules in OWL

To get a consistency check, must violate one of the model rules
- sameAs/differentFrom assertions
- member of disjoint classes
- etc.

What about general data conformance rules?
- OWL does not support this
- put another way: you must find a way to define your data such that an inconsistency arises
- example: string for propertyX must be in nnn-nn-nnnn form
  - define a local restriction for classX on propertyX using pattern datatype
  - define a class that is the converse of all classes with this restriction (i.e. all members of the class not in nnn-nn-nnnn form)
  - Prove that some datum is in this class and its inverse
Consistency Checking vs. Data Constraints

- SPIN offers a more flexible alternative
  - constraint violations run separately from rules (inference)
    - run rules to infer
    - constraint violation warnings for data constraints

- Constraint checking is independent of inference
  - from previous example
    - define a constraint rule a local restriction for classX on propertyX using pattern datatype (e.g. using ASK queries)
    - (better yet, just define the rule)
    - run constraint violations
  - data constraints do not need to follow inferencing rules
Kinds of Inference

- Inference
  - OWL-DL performs FOL reasoning
  - many can inferences be interpreted as rule-based inferences

- Consistency checking
  - semantics designed to maintain consistency
  - will typically add triples to accomplish this
  - example with FunctionalProperty
  - OWL defines these consistency checks

```owl
CONSTRUCT {
}
WHERE {
  ?person a :Person .
  ?person :birthYear ?birthYear .
}
```
Inference and Consistency

- RDFS and OWL have a notion of what it means to create a consistent model
  - based on FOPL
  - to make a model consistent, insert the missing fact
    - in RDF, a triple is a fact
  - if the fact cannot be inserted, then the model is inconsistent

- Using owl:FunctionalProperty as an example
  - can be used to infer that two individuals are the same
  - ...unless there are facts that contradict this
If a property is functional, then there can be only one individual related

- if there are two individuals related by the property then can assume the individuals are the same

birthFather example

- birthFatherrdf:typeowl:FunctionalProperty
- TedKennedyimplies Ted and Edward are the same individual
- EdwardKennedy
Inference with Functional Properties

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>TedKennedy</td>
<td>profession</td>
<td>senator</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>lastName</td>
<td>Kennedy</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>parent</td>
<td>RoseFitzgerald</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>almaMater</td>
<td>University</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>child</td>
<td>KaraKennedy</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>name</td>
<td>Edward Kennedy</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>middleInitial</td>
<td>Moore</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>photo</td>
<td>Virginia</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>spouse</td>
<td>VictoriaReggie</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>birthYear</td>
<td>1932</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>profession</td>
<td>army</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>gender</td>
<td>male</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>child</td>
<td>PatrickKennedy</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>almaMater</td>
<td>Harvard</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>parent</td>
<td>JosephKennedy</td>
</tr>
<tr>
<td>TedKennedy</td>
<td></td>
<td>Edward</td>
</tr>
<tr>
<td>TedKennedy</td>
<td>firstName</td>
<td></td>
</tr>
<tr>
<td>TedKennedy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VictoriaGifford</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Inferred** ‘TedKennedy owl:sameAs EdwardKennedy’ and inferred all facts we know about EdwardKennedy.
An Brief Introduction to Open-World Reasoning

- Functional property assumption in open world reasoning
  - if there are two functional properties, then instances are the same!
  - “close the world” by stating a fact about the instances

- OWL/RDFS semantics are designed to maintain consistency
  - by defining an inference that ensures the consistency
  - but what if there is an inconsistency?
  - a second type of reasoning – consistency checking...
What if Ted and Edward aren’t the same
- can assert this as a new fact:
  
  :TedKennedy owl:differentFrom :EdwardKennedy
- now there is an inconsistency
owl:differentFrom

- Run inferences again
  - verify that the inferences were still made
  - i.e. ‘Ted owl:sameAs Edward’ and Ted ‘owl:differentFrom Edward’

- But isn’t there an error here?
  - i.e. we defined birthFather as a functional property
    - ...there can only be one
  - partial answer: it’s an inconsistent definition, not an error

- Configure to use TopSPIN and OWL 2 rules
  - TopSPIN will perform the consistency checks
  - ...OWLIM does not
Closed-World Reasoning (2)

- `owl:FunctionalProperty` example

Open world interpretation of functional property: if two properties exist for a functional ?s ?p triple, then ?o are the same

```
# prp-fp
CONSTRUCT {
    ?y1 owl:sameAs ?y2 .
}
WHERE {
    ?p a owl:FunctionalProperty .
    FILTER (?y1 != ?y2) .
}
```

Closed world interpretation: if two properties exist for a functional ?s ?p triple, then there is a problem with the data

```
# prp-fp
CONSTRUCT {
    _:b0 a spin:ConstraintViolation .
    _:b0 spin:violationRoot ?this .
    _:b0 spin:violationPath ?p .
    _:b0 rdfs:label "owl:functionalProperty constraint" .
}
WHERE {
    ?p a owl:FunctionalProperty .
    FILTER (?y1 != ?y2) .
}
```
Run inferences and display constraint violation warnings
SPIN Constraints

- Constraints are used to check data
  - if a constraint is violated, flag a warning
  - as with rules, attach to class and apply to all members of the class
  - must turn on warnings

spin:constraint for :Person

# age must be within a realistic range
ASK WHERE
{  ?this :age ?age .
    FILTER ((?age < 0) || (?age > 102)) .
}

- Note this is an ASK query
- If it returns true, the constraint has been violated
Warnings are displayed on applied resources

(changed Jean Kennedy’s birth date)

Either kennedys:spouse or kennedys:gender could be the problem
Use `spin:ConstraintViolation` to target where warning is placed
- use `CONSTRUCT` to build constraint warning triples

```
CONSTRUCT {
  _:b0 a spin:ConstraintViolation .
  _:b0 spin:violationRoot ?this .
  _:b0 spin:violationPath kennedys:spouse .
  _:b0 rdfs:label "Always blame it on the spouse".
}
WHERE {
  FILTER (?gender = ?spouseGender).
}
```

Gender no longer displays a warning
Reasoning and OWL Inference
OWL 2 Profiles
- profiles are subsets of the full OWL 2 spec
- defined to maximize reasoning efficiency
  - ...through restrictions of full language

Profiles
- OWL 2 EL (TBox reasoning – classes, properties)
- OWL 2 RL (ABox reasoning – instances)
- OWL 2 QL (Query answering)
OWL 2 RL Profile

- OWL 2 RL: forward-chaining rule systems
  - similar to principles followed in Description Logic Programs (DLP) and pD*
  - reasoning defined as a set of rules
  - restrictions on OWL 2 to ensure that the reasoner reasons only with individuals that occur explicitly in the ontology

- Profile Highlights:
  - most OWL 2 constructs are allowed
    - some exceptions, such as reasoning over datatypes
  - “OWL 2 RL supports all axioms of OWL 2 apart from disjoint unions of classes (DisjointUnion) and reflexive object property axioms (ReflexiveObjectProperty).” OWL 2 Web Ontology Language Profiles [http://www.w3.org/TR/owl2-profiles/#OWL_2_RL]
Partial axiomatization of the OWL 2 RDF-Based Semantics

- first-order implications, specified in an abstract RDF syntax

### Table 5. The Semantics of Axioms about Properties

<table>
<thead>
<tr>
<th></th>
<th>If</th>
<th>then</th>
</tr>
</thead>
<tbody>
<tr>
<td>prp-ap</td>
<td>$T(p, rdfs:domain, ?c)$&lt;br&gt;$T(x, p, ?y)$</td>
<td>$T(x, rdf:type, ?c)$</td>
</tr>
<tr>
<td>prp-dom</td>
<td>$T(x, rdf:type, ?c)$</td>
<td>$T(x, rdf:type, ?c)$</td>
</tr>
<tr>
<td>prp-rng</td>
<td>$T(x, p, ?y)$</td>
<td>$T(y, rdf:type, ?c)$</td>
</tr>
<tr>
<td>prp-fp</td>
<td>$T(x, p, rdf:type, owl:FunctionalProperty)$</td>
<td>$T(y_1, owl:sameAs, y_2)$</td>
</tr>
<tr>
<td>prp-ifp</td>
<td>$T(x, p, rdf:type, owl:InverseFunctionalProperty)$&lt;br&gt;$T(x, p, ?y)$</td>
<td>$T(x_1, owl:sameAs, x_2)$</td>
</tr>
<tr>
<td>cls-svf2</td>
<td>$T(x, owl:someValuesFrom, owl:Thing)$&lt;br&gt;$T(x, owl:onProperty, p)$&lt;br&gt;$T(u, p, v)$</td>
<td>$T(u, rdf:type, ?v)$</td>
</tr>
<tr>
<td>cls-avf</td>
<td>$T(x, owl:allValuesFrom, ?y)$&lt;br&gt;$T(x, owl:onProperty, p)$&lt;br&gt;$T(u, rdf:type, ?x)$&lt;br&gt;$T(u, p, v)$</td>
<td>$T(v, rdf:type, ?u)$</td>
</tr>
</tbody>
</table>

```owl
# prp-fp
CONSTRUCT { ?y1 owl:sameAs ?y2 }
WHERE {
    ?p a owl:FunctionalProperty .
    FILTER (?y1 != ?y2).
}

# cls-svf2
CONSTRUCT { ?u a ?x .
WHERE {
    ?x owl:someValuesFrom owl:Thing .
    FILTER (?u != ?v).
}
```
Many OWL reasoners started with some form of rule-based reasoning
- standard defines OWL inferences in rule form for RDF
- ...both inferences and constraints

**Table 7 specifies the semantic conditions on class axioms.**

**SPIN Constraint:**

```
# cax-dw
CONSTRUCT {
  _:b0 a spin:ConstraintViolation .
  _:b0 spin:violationRoot ?x .
  _:b0 rdfs:label "Shared instance of disjoint classes" .
}
WHERE {
  ?x a ?c1 .
  ?x a ?c2 .
}
```
OWL 2 in TBS

- Two steps to run OWL inferences
  - create a SPARQL Rules (SPIN) file
    - this sets reasoner to TopSPIN (Inference > Configure Inferencing)
  - import OWL rules
    - in model home choose OWL/RDFS+
    - or import: TopBraid > owlrl-all.rdf
OWL 2 Rules in SPARQL

- owlrl.rdf creates a template for each rule from the OWL2 RL profile
  - TopSPIN executes rules
  - forward chaining creates all entailments

- How can you control OWL inferences?
  - reasoners take an all-or-none approach
  - RL is a set of rules – delete what is not needed
  - Hint: remove eq-ref, it’s useless

Can mix and match these as desired – with caveats on the OWL notion of consistent models
Creating Rules from OWL Standard

- Suppose you need to implement some subset of OWL
  - not interested in all rules
  - interested in a subset, but want standard rules
  - for all RDFS/OWL reasoners: can’t do that
    - no way to pick-and-choose rules
- In SPIN, import owlrl.owl
  - create rules from templates

OWL RL templates, indexed by names in recommendation

SPARQL implementation of rule
Using OWL and SPARQL CONSTRUCT

- SPARQL rules and OWL can be used together
  - one single syntax – SPARQL
- Example: user-defined datatypes
  - RL profile designed conservatively – some constructs left out for possible computational issues
  - can be introduced as SPARQL rules

**Example: OWL restriction for max cardinality**

Person and (hasAge only xsd:integer[>= 21])

```sparql
#OWL rule for max cardinality
CONSTRUCT {
  ?x a ?restriction .
}
WHERE {
  ?datatype owl:onDatatype xsd:integer .
  ?datatype owl:withRestrictions ?var .
  ?datatype a rdfs:Datatype .
  ?restriction owl:allValuesFrom ?datatype .
  ?restriction a owl:Restriction .
  ?var1 xsd:minExclusive ?mval .
  FILTER (?val > ?mval) .
}
```
Inferencing in the Good Relations Ontology
Good Relations Ontology

- Developed by Martin Hepp at the Universitaet der Bundeswehr, Muenchen
- Lets businesses describe their goods
- Used by Google, Yahoo!, etc. for SEO

Hepp Research Personal SCSI Controller Card
The Hepp Research Personal SCSI is a 16-bit add-on card that allows attaching up to seven SCSI devices to...
www.heppresearch.com/commercecollator - Cached - Similar
Good Relations Example -

- Product Description for Plush Beauty Bar
- A business entity (that’s the boutique itself) offers “NAIL SERVICES” which includes a single manicure for $19
- How many manicures do you get? (assume one)
- A one-and-a-half hour massage costs $80
- This is a different representation (reified to talk about the price and the timing)
Explicitly representing “One Manicure”
Consistency in Good Relations

Issue: How can Good Relations let users specify “Plush Beauty Bar offers a Manicure for $19”, but have it be consistent with representations that say “A one-and-a-half hour massage costs $80”?

Solution 1: Make everyone reify all their offerings the same way

Solution 2: Have a rule that relates the two together.
# Expand gr:includes between gr:Offering and grProductOrServiceModel (newer variant)

**CONSTRUCT** {
  ?o gr:includesObject _:b0 .
  _:b0 a gr:TypeAndQuantityNode .
  _:b0 gr:amountOfThisGood "1.0"^^xsd:float .
  _:b0 gr:hasUnitOfMeasurement "C62"^^xsd:string .
  _:b0 gr:typeOfGood _:b1 .
  _:b1 a gr:ProductOrService .
  _:b1 rdfs:label ?label.
}

**WHERE** {
  ?o a gr:Offering .
  NOT EXISTS {
    ?o gr:includesObject ?tqn .
  }
}
What can’t SPARQL do?

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure

owl:Restriction

owl:equivalentClass

EuropeanCapitalHoliday

rdf:type

Disney’s Euro Adventure
### OWL + SPARQL

<table>
<thead>
<tr>
<th>OWL</th>
<th>SPARQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic structure</td>
<td>Query processing</td>
</tr>
<tr>
<td>Open-World</td>
<td>Closed-World</td>
</tr>
<tr>
<td>T-Box (class-level reasoning)</td>
<td>A-box (data-level) reasoning</td>
</tr>
<tr>
<td>Classifications</td>
<td>Transformations</td>
</tr>
<tr>
<td>Consistency</td>
<td>Constraints</td>
</tr>
<tr>
<td>“Solution” processing – find a solution to many simultaneous constraints</td>
<td>“Waterfall” processing – take what you know and learn something new</td>
</tr>
</tbody>
</table>
Reasoning with SPARQL Update
Modifying Data with RDFS and OWL

- OWL & RDFS can add new data, but cannot remove data
  - monotonic languages
  - facts can be asserted/inferred, but not retracted

- Replace one value of properties with another
  - e.g. :label used in kennedys model
  - ...want to convert to using rdfs:label
  - can easily add in OWL via subPropertyOf
    - ...but cannot remove old values

Two issues:
1) inferred vs. asserted
2) only want the value once
Data Modification with SPARQL

- **SPARQL CONSTRUCT** adds triples
  - how to remove triples?
  - need some logic outside of SPARQL
  - SPARQLMotion as an example

**Example SPARQL Statements**

```
CONSTRUCT
{
  ?s rdfs:label ?name .
}
WHERE
{ ?s kennedys:name ?name . }

sml:replace = true
```

```
CONSTRUCT
{ ?s kennedys:name ?name .
}
WHERE
{ ?s kennedys:name ?name . }

sml:replace = true
```

Only triples created, e.g.
{ kennedys:JeanKennedy rdfs:label "JeanKennedy" }
Data Modification with SPARQL Update

- SPARQL INSERT/DELETE to update data
  - syntax is the same as CONSTRUCT
    - ...specify a graph of RDF triples
  - but instead of returning triples, the specified graph(s) is inserted/deleted

```
INSERT {?s rdfs:label ?name } WHERE { ?s kennedys:name ?name }
```

**INSERT the triples directly into the current graph**

```
DELETE {?s kennedys:name ?name } WHERE { ?s kennedys:name ?name }
```

**DELETE the triples directly from the current graph**

```
INSERT { GRAPH <http://topbraid.org/examples/kennedys> {?s rdfs:label ?name } }
WHERE { ?s kennedys:name ?name }
```

To insert into a specific graph, use the GRAPH keyword.
WITH keyword specifies a keyword for modifying data

- other syntax for named graph access

WITH <http://topbraid.org/examples/kennedys>
DELETE {?s kennedys:name ?name } 
INSERT {?s rdfs:label ?name } 
WHERE { 
    ?s kennedys:name ?name 
}

DELETE {?s :count ?oldcount . } 
INSERT {?s :count ?new . } 
WHERE { 
    ?s :count ?oldcount . 
    BIND ((?oldcount + 1) AS ?new) 
}

DELETE {?s :lastEditDate ?old . } 
INSERT {?s :lastEditDate ?new . } 
WHERE { 
    ?s :lastEditDate ?old . 
    BIND (afn:now() AS ?new) 
}
Update broadens the context of Semantic Web reasoning:
- both infer and retract triples
- SPIN is an example reasoner using SPARQL Update

There are details:
define spin:updateRule as a subproperty of spin:rule and define max iterations to some value

for each member of kennedys:Person (and subclasses thereof), create a string in the form “firstName, lastName”, insert that as name and delete the old value.
PerformUpdate module applies an update query
  - multiple queries per module
  - can order updates through sm:next