Summary of RDFS-Plus

A subset of OWL constructs, called RDFS-Plus, is singled out for presentation here for several reasons. Pedagogically, these constructs are a gentle addition to those already familiar from RDFS; practically, this set of constructs has considerable utility; and, computationally, this OWL subset can be implemented using a wide variety of inferencing technologies. RDFS-Plus (like RDFS) is expressed entirely in RDF. The namespace is `owl:`. As with RDFS, the meanings of constructs are specified by the rules governing inferences made with them, and the actions of an inference engine combine features of the schema language in novel ways.

Inverse

The inverse of a property *P* is another property *Q* that reverses its direction. The rule is

Given *P* `owl:inverseOf` *Q* and *x P y*, infer *y Q x*

Challenge: Integrating Data that Do Not Want to be Integrated

In the Property Union challenge, we had two properties, `borrows` and checkedOut, and combined them under a single property by making them both sub-properties of `hasPossession`. We were lucky the two data sources had the same domain (`Patron`) and range (`Book`). But suppose the domain and range of the second data source are reversed: for each book, there is a field for the patron the book is `signedTo`.

Challenge 10

Merge `signedTo` and `borrows` as we merged `borrows` and `checkedOut`.

Solution

Use `owl:inverseOf` so the domains and ranges do match and define a new property as the inverse of `signedTo`:

```
:signedTo  owl:inverseOf  :signedOut .
```

Now use the original Property Union pattern to merge `signedOut` and `borrows`.

Solution (Alternative)

The choice to specify an inverse for `signedTo` rather than for `hasPossession` was somewhat arbitrary. We can use the same rules for `owl:inverseOf` and `rdfs:subPropertyOf` in a different order yet resulting in the same `hasPossession` triples:

```
:signedTo  rdfs:subPropertyOf  :possessedBy .
:borrows  rdfs:subPropertyOf  :hasPossession .
:possessedBy  owl:inverseOf  :hasPossession .
```

Since we want not only to support desired inferences but also to support reuse, we should specify inverses for every property, regardless of `subProperty` level.

Challenge: Using the Modeling Language to Extend the Modeling Language

Extra OWL features beginners find missing often can be supported by OWL as a combination of other features.

Challenge 11

Define a `superClassOf` property so the parent class is the subject of all definitional triples. (With `rdfs:subClassOf`, all triples would go the other way.) Use namespace `myowl:` for this relation.

Solution

Do this simply by declaring an inverse:

```
myowl:superClassOf  owl:inverse  rdfs:subClassOf .
```

Beginners often think they have no right to extend the meaning of the OWL language or to make statements about the OWL and RDFS resources (e.g., `rdfs:subClassOf`).

Challenge: The Marriage of Shakespeare

Earlier, we could assert that Anne Hathaway married Shakespeare but could not assert that Shakespeare married Anne

Challenge 12

Infer marriages in the reverse direction from that in which they are asserted.

Solution

Just declare `bio:married` to be its own inverse.

This patter of self-inverses is so common that it is built into OWL using the special construct `owl:SymmetricProperty`.

Symmetric Properties

Unlike `owl:inverseOf` (a property relating two other properties), `owl:SymmetricProperty` is an aspect of a single property, expressed as a class:

```
P  a  owl:SymmetricProperty .
```

From this, infer

```
P  owl:inverseOf  P .
```

Using OWL to Extend OWL

When we merged the inverse of `signedTo` (viz., `signedOut`) with borrow (in Challenge 10), the inverse relations went in only one direction. We want these relations to go in both directions:

```
:hasPossession  owl:inverseOf  :possessedBy .
:signedOut  owl:inverseOf  :signedTo .
:borrows  owl:inverseOf  :lentTo .
```

Challenge 13

Infer these triples without having to assert them.

Solution

```
owl:inverseOf  a  owl:SymmetricProperty .
```

Transitivity

In mathematics, relation *R* is transitive if *R(a,b)* and *R(b,c)* implies *R(a,c)*. Like `owl:SymmetricProperty`, `owl:TransitiveProperty` is a class of properties:

```
P  a  owl:TransitiveProperty .
```

The rule is:

Given `P a owl:TransitiveProperty`, `x P y`, and `y P z`, infer x P z.

Using transitivity repeatedly, we can pull together the start and end individuals of a chain of individuals related by a transitive property `P`:

Given x_0 P x_1, x_1 P x_2, …, and x_{n-1} P x_n, infer x_0 P x_n.

Typical examples of transitive properties include

- Ancestor/descendant: If x is an ancestor of y and y is an ancestor of z then x is an ancestor of z.
- Geographical containment: If Tokyo is in Japan and Japan is in Asia, then Tokyo is in Asia

Challenge: Relating Parents to Ancestors

A model of genealogy includes parents and ancestors.

Challenge 14

Allow a model to maintain consistent ancestry information, given parentage information.

Solution

First define the parent property to be a `subPropertyOf` the ancestor property, then declare ancestor (only) to be transitive.

Challenge: Layers of Relationships

It is sometimes controversial whether a property is transitive. For example, Favre's thumb is part of Favre and Favre is part of the Vikings, but … To anticipate possible uses of the model, we should support both points of view when controversy might arise.

Challenge 15

Simultaneously maintain transitive and non-transitive versions of part-of information.

Solution

Define two versions of the `partOf` property in different namespaces (or with different names) with one a `subPropertyOf` the other and the super-property declared transitive

Managing Networks of Dependencies

Now see how the constructs `rdfs:subPropertyOf`, `owl:inverseOf`, and `owl:transitiveProperty` can be combined to model aspects of networks of dependencies. In workflow management, for example, the sequence of tasks in a complex working situation is represented explicitly, and the progress of each task is tracked in that sequence. Workflow is modeled in a Semantic Web so that parts of it can be shared with others, encouraging reuse, review, and publication of work fragments

We take a simple example: making ice cream. Property `dependsOn` represents dependencies between steps. Its inverse is `enables`.

Challenge 16

For a given step, we want to know all the steps it depends on or all that depend on it

Solution

Use the `subPropertyOf`/`TransitiveProperty` pattern for both `dependsOn` and `enables`:

```
:dependsOn rdfs:subPropertyOf :hasPrerequisite .
:hasPrerequisite  a  owl:TransitiveProperty .
:enables rdfs:subPropertyOf :prerequisiteFor .
:prerequisiteFor  a  owl:TransitiveProperty .
```

Challenge 17

In a realistic setting, we would manage several interacting processes and might even lose track of what steps are in the same procedure. In the recipe example, can we connect the steps in the same recipe together?

Solution

Combine both fundamental relationships (`enables` and `dependsOn`) as common sub-properties of `neighborStep`, and make `neighborStep` a sub-property of a transitive property, `inSameRecipe`. Any two steps here will be related by inSameRecipe. This includes relating each step to itself, which might not be what we want.

Challenge 18

Define a property that relates a recipe only to the other steps in the same recipe.

Solution

We earlier defined properties `hasPrerequisite` (looking downstream along the dependencies) and `prerequisiteFor` (looking upstream). Now join these under a common super-property, `otherStep`, that is not transitive. For `otherStep`, the set of triples involving, e.g., `GraduallyMix` is just the union of the sets for `hasPrerequisite` and `prerequisiteFor`. There is no reflexive triple.

For `inSameRecipe`, a single transitive property is at the top of the subproperty tree, and both primitive properties are brought together. Any combinations of the resulting property (`neighborStep`) are chained together as a transitive property (`inSameRecipe`). For `otherStep`, in contrast, the top property itself is not transitive. It is a simple combination of two transitive properties; inference for each is done separately and the results combined.

Equivalence

In RDF, the URI provides global identity, valid across data sources, allowing us to refer to a single entity in a distributed way. But suppose we have information from multiple sources controlled by multiple stakeholders. Any two stakeholders might use different URIs to refer to the same entity. In a federated setting, we want to stipulate that two URIs actually refer to the same entity.

Equivalent Classes

We have used the double-`subClassOf` idiom to express that one class, *A*, has the same elements as another, *B*. (When 2 classes are known always to have the same members, they are *equivalent*.) RDFS-Plus provides a simpler way to express equivalence of classes:

A `owl:equivalentClass` *B* .

For classes *A* and *B* and individual *x*, we have two rules:

Given *A* `owl:equivalentClass` *B* and *x* `a` *A*, infer *x* a *B*.

Given *B* `owl:equivalentClass` *A* and *x* `a` *B*, infer *x* `a` *A*.

But the following is given that

`owl:equivalentClass a owl:SymmetricProperty .`

So we don't need the second rule—infer it from the first. We do not even need the first rule since it's also given that

`owl:equivalentClass rdfs:subPropertyOf rdfs:subClassOf .`

So, given equivalence, we get the double-`subClassOf` characterization from what is built in.

Equivalence means only that the twoclasses have the same members. Other properties of the classes (e.g., `rdfs:label` values) are not shared.

Equivalent Properties

We have used the double-`subPropertyOf` idiom to express that one property, *P*, is equivalent to another, *Q*. RDFS-Plus also has a simpler way to express property equivalence:

P `owl:equivalentProperty` *Q* .

For properties *P* and *Q* ad individuals *x* and *y*, we again have two rules:

Given *P* `owl:equivalentProperty` *Q* and *x P y*, infer *x Q y*.

Given *P* `owl:equivalentProperty` *Q* and *x Q y*, infer *x P y*.

But, again, these rules can be derived from what is built in:

```
owl:equivalentProperty rdfs:subPropertyOf owl:subPropertyOf .
owl:equivalentProperty a owl:SymmetricProperty .
```

Rather than just noticing that the rule governing `owl:equivalentProperty` is the same as that governing `rdfs:subPropertyOf` (except it works both ways), we actually express these facts with triples. By making `owl:equivalentProperty` a sub-property of `rdfs:subPropertyOf`, we explicitly assert that they are governed by the same rule. By making `owl:equivalentProperty` an `owl:SymmetricProperty`, we assert that this rule works in both directions. This makes the relationship between the parts of the OWL language explicit and, in fact, models them in OWL.

Same Individual

`owl:equivalentClass` and `owl:equivalentProperty` provide simple ways to express what is already expressible in RDFS—they do not increase the expressive power of RDFS-Plus.

Now, things in the world, members of classes, are called *individuals* (i.e., real-word resources). We have seen surgeons, plays, countries, playwrights, … In the absence of agreement on global names, different Web authors select different URIs for the same individual. When information from the different sources is brought together in the distributed network of data, the Web infrastructure has no way to know that certain pairs of names should be treated as denoting the same individual. The flip side is that we cannot assume that, just because two URIs are distinct, they denote distinct individuals. The *Non-unique Naming Assumption*: We must assume (until told otherwise) that some Web resource might be referred to using different names by different people.

When we do determine that two individuals originally considered separately are in fact the same individual, we use the `owl:sameAs` construct, whose meaning is given by the rule

Given *A* `owl:sameAs` *B* and a triple containing *A*, infer a triple that is identical except that *A* is replace by *B*.

We also have that `owl:sameAs` is symmetric.

Challenge: Merging Data from Different Databases

Recall how to interpret info in a table as RDF triples. This example involves a table, `Product`, that translates into 63 triples for the seven columns and nine rows. The following is an example triple relating to `Manufacture_Location`.

```
mfg:Product1 mfg:Product_Manufacture_Location "Sacramento" .
```

Suppose another division in the company keeps a table, also called `Product`, of the facilities needed to produce various parts. Some of the products in the first table appear in the second table and some do not.

Challenge 19

Using the products in both tables, write a federated query cross-referencing cities with the facilities required for the production there.

Solution

If the tables were in a single database, there could be foreign-key references from one table to the other, and we could join the two tables. But the tables are from different databases, so there is no such common reference.

When we turn the tables into triples, the individuals corresponding to rows are assigned global identifiers. Use namespace `p:` for the second table. The following is an example triple corresponding to a required facility.

```
p:Product1 p:Product_Facility "Assembly Center" .
```

The global identifiers aren't the same, so we need

```
p:Product1 owl:sameAs mfg:Product4 .
```

We can now match the SPARQL pattern

```
[?p p:Product_Facility ?facility .
 ?p mfg:Product_Manufacture_Location ?location .]
```

and display `?facility` and `?location`.

The solution relied on us knowing which product in one table matched with which in the other. In a real situation, the data in the tables change frequently, and it is not practical to assert all the `owl:sameAs` triples by hand. The solution to this problem follows.

Computing Sameness—Functional Properties

Functional Properties

A functional property *P* can take only one value for any given individual. The following is the rule (where *A* and *B* are resources).

Given *P* `a owl:FunctionalProperty`, *x P A*, and *x P B*, infer *A* `owl:sameAs` *B*.

Functional properties allow sameness to be inferred, but, for determining sameness, `owl:InverseFunctionalProperty` is much more common than `owl:FunctionalProperty`.

Inverse Functional Properties

Some consider `owl:InverseFunctionalProperty` the most important modeling construct in RDFS-Plus. Think of `owl:InverseFunctionalProperty` as the inverse of `owl:FunctionalProperty`. The following is the rule (where again *A* and *B* are resources).

Given *P* `a owl:InverseFunctionalProperty`, *A P x*, and *B P x*, infer *A* `owl:sameAs` *B*.

An `owl:InverseFunctionalProperty` plays the role of a key field in a relational database. Unlike with a relational database, however, RDFS-Plus does not signal an error if two entities are found to share a value for an inverse functional property. Instead, it infers that the two are the same. Any identifying number (e.g., SSN, driver's license number, …) is an inverse functional property.

Challenge 20

Infer the appropriate `owl:sameAs` triples from the data already asserted regarding the two tables discussed above.

Solution

We want to find an inverse functional property present in both data sets and use it to bridge them. The two tables both have a field called `ModelNo`, and, if two products have the same model number, they are the same product, so

```
mfg:Product_ModelNo  a  owl:InverseFunctionalProperty .
```

We have to arrange that we are talking about a single property:

```
p:Product_ModelNo owl:equivalentProperty mfg:Product_ModelNo .
```

We could do these steps in either order, and we could write the last triple with subject and object reversed.

Most real data integration situations rely on more elaborate notions of identity that include multiple properties and encounter uncertain or ambiguous cases. This problem can often be solved with combinations of OWL properties we explore later. A fully general solution remains a research topic

Combining Functional and Inverse Functional Properties

It is often useful for a property to be one-to-one, that is, both functional and inverse functional. This is often the case for identification numbers.

Challenge 21

Assign to students id numbers used for billing and assigning grades. No two students should share an id number, and no student should have more than one id number.

Solution

Define property `hasIdenityNo` with the appropriate domain and range, and enforce uniqueness by asserting

```
:hasIdentityNo  a  owl:FunctionalProperty .
:hasIdentityNo  a  owl:InverseFunctionalProperty .
```

More RDFS-Plus Constructs

The small extensions RDFS-Plus provides greatly increase the applicability of RDFS. `owl:inverseOf` combines with `rdfs:subClassOf` to let us align properties that are not expressed in compatible ways in existing data schemas, and `owl:TransitiveProperty` combines with `rdfs:subPropertyOf` in several novel ways, letting us model various relationships among chains of individuals. From a Semantic Web perspective, the most applicable extensions are those dealing with sameness: `sameAs`, `FunctionalProperty`, `InverseFunctionalProperty`. They provide a way to describe how information from multiple sources is to be merged in a distributed web of information.

Classes `owl:DataTypeProperty` and `owl:ObjectProperty` let us distinguish between properties whose values are objects and those whose values belong to some datatype. They provide no additional semantics, but they do provide useful discipline as well as information for editing tools for, e.g., displaying models. Most OWL tools prefer to distinguish datatype properties (e.g., `ship:maidenVoyage`) and object properties (e.g., `bio:married`).

Another distinction OWL makes is between `rdfs:Class` and `owl:Class`. Since OWL is based on RDFS, backward compatibility is not an issue, and the OWL specification stipulates

```
owl:Class rdfs:subClassOf rdfs:Class .
```

Most tools require that classes used in OWL models be declared members of `owl:Class`, but most model editors declare a class an `owl:class` automatically when it is created. We discuss some subtle distinctions later.