7 OWL-Fast:

RDFS provides a very limited set of inference capabilities which, as we have seen, have considerable utility in a semantic web setting for merging information from multiple sources. In this chapter, we take the first step toward the Web Ontology Language OWL, in which much more elaborate constraints on how information is to be merged can be specified. We have selected a particular set of OWL constructs to present at this stage. This set was selected to satisfy a number of goals:

- Pedagogically, these constructs constitute a gentle addition to the constructs that are already familiar from RDFS, increasing the power of the language without making a large conceptual leap from RDFS,
- Practically, we have found that this set of OWL constructs has considerable utility in the information integration projects we have done. In fact, it is much easier to find and describe case studies using RDFS plus this set of OWL constructs than it is to find case studies that use RDFS on its own,
- Computationally, this subset of OWL can be implemented using a wide variety of inferencing technologies, lessening the dependency between the Semantic Web and any particular technology.

For these reasons, we feel that this particular subset will have value beyond the pedagogical value in this book. We call this subset of OWL *OWL-Fast*, because we see a trend amongst vendors of Semantic Web tools of determining a subset of OWL that is at the same time useful and can be implemented quickly. We have identified this particular subset via an informal poll amongst cutting-edge vendors and from our own experience with early adopters of Semantic Web technology.

Just as was the case for RDFS, OWL-Fast is expressed entirely in RDF. The only distinction is that there are a number of resources, all in the namespace *owl:* as described in section **Error! Reference source not found.**. The meaning of these resources is specified by the rules that govern the inferences that can be made from them.

In the case of RDFS, we saw how the actions of an inference engine could be used to combine various features of the schema language in novel ways. This trend will continue for OWL-Fast, but as you might expect, the more constructs we have to begin with, the more opportunity we have for useful and novel combinations.

7.1 Inverse

The names of many of the OWL constructs come from corresponding names in mathematics. Despite their mathematical names, they also have a more common, everyday interpretation. The idea *owl:inverseOf* is a prime example; if a relationship, say *hasParent* is interesting enough to mention in a model, then it's a good bet that another relationship, say *hasChild*, is also interesting. Because of the evocative names *hasParent* and *hasChild*, you can guess the relationship between them. The OWL construct *owl:inverseOf* makes the relationship between *hasParent* and *hasChild* explicit, and describes precisely exactly what it means.

In mathematics, the inverse of a function f (usually written as f^{-1}) is the function that satisfies the property that if $f(x)=y$, then the $f^{-1}(y)=x$. Similarly in OWL, the inverse of a property is another property that reverses its direction.

To be specific, we look at the formal meaning of *owl:inverseOf*. In OWL, as in RDFS, the meaning of any construct is given by the inferences that can be drawn from it. If we have the following triples

```
P owl:inverseOf Q .
x P y .
```

Then we can infer that:

```
y Q x .
```

In the examples in the book, we have already seen a number of possibilities for inverses, even though we haven't used them so far. In our Shakespeare examples, we have the triples

```
lit:Shakespeare lit:wrote lit:Macbeth .
lit:Macbeth lit:setIn geo:Scotland .
```

If, in addition to these triples, we also state some inverses, e.g.,

```
lit:wrote owl:inverseOf lit:writtenBy .
lit:settingFor owl:inverseOf lit:setIn .
```

then we can infer that

```
lit:Macbeth lit:writtenBy lit:Shakespeare .
geo:Scotland lit:setingFor lit:Macbeth .
```

While the meaning of *owl:inverseOf* is not difficult to describe, what is the utility of such a construct in a modeling language? After all, the effect of *inverseOf* can be achieved just as easily by writing the query differently. For instance, if we want to know all the plays that are *setIn* Scotland, we can use the inverse property *settingFor* in our query pattern, e.g.,

```
{geo:Scotland lit:settingfor ?play . }
```

Because of the semantics of the inverse property, this will give us all plays that were *setIn* Scotland.

But we could have avoided the use of the inverse property, and simply written the query as

```
    {?play lit:setIn geo:Scotland . }
```

We get the same answers, and we don't have any need for an extra construct in the modeling language.

While this is true, *owl:inverseOf* nevertheless does have considerable utility in modeling, based on how it can interact with other modeling constructs. In the next challenge, we'll see how the Property Union challenge from section **Error! Reference source not found.** can be extended using inverses.

7.1.1 Challenge: Integrating data that doesn't want to be integrated

In the Property Union challenge from section **Error! Reference source not found.**, we had two properties, *borrows* and *checkedOut*.We were able to combine them under a single property by making them both *rdfs:subPropertyOf* the same parent property, *hasPosession*. We were fortunate that the two sources of data happened to link a *Patron* as the subject to a *Book* as the object (i.e., they had the same domain and range). Suppose instead that the second source was instead an index of books, and for each book there was a field specifying the patron the book was *signedTo* (i.e., the domain and range are reversed).

CHALLENGE

How can we merge *signedTo* and *borrows*, in a way analogous to how we merged *borrows* and *checkedOut*, given that *signedTo* and *borrows* don't share domains and ranges?

SOLUTION

The solution involves a simple use of *owl:inverseOf* to specify two properties for which the domain and range do match, as required for the merge. We define a new property, say *signedOut*, as the inverse of *signedTo*, as follows:

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

Now we can use the original Property Union pattern to merge signedOut and borrows into the single hasPossession property:

```
signedOut rdfs:subPropertyOf hasPossession .
borrows rdfs:subPropertyOf hasPossession .
```

So, if we have some data expressed using signedTo, along with data expressed with borrows, as follows:

```
Amit borrows MobyDick .
Marie borrows Orlando .
LeavesOfGrass signedTo Jim .
WutheringHeights signedTo Yoshi .
```

Then with the rule for *inverseOf*, we have the additional triples

```
Jim signedOut LeavesOfGrass .
Yoshi signedOut WutheringHeights.
```

and with subPropertyOf, we have

```
Amit hasPossession MobyDick .
Marie hasPossession Orlando .
Jim hasPossession LeavesOfGrass .
Yoshi hasPossession WutheringHeights.
```

as desired.

SOLUTION (alternative)

There is a certain asymmetry in this solution; the choice to specify an inverse for *signedTo* rather than for *hasPossession* was somewhat arbitrary. Another solution, also utilizing *owl:inverseOf* and *rdfs:subPropertyOf* and just as viable as the first, is as follows:

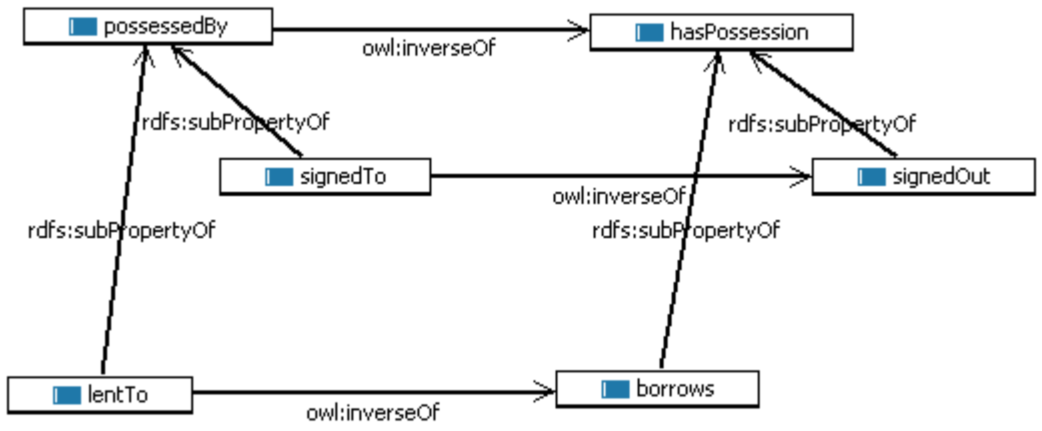
```
signedTo rdfs:subPropertyOf possessedBy .
borrows rdfs:subPropertyOf hasPossession .
possessedBy owl:inverseOf hasPossession .
```

These statements use the same rules for *owl:inverseOf* and *rdfs:subPropertyOf*, but in a different order, resulting in the same *hasPossession* triples. Which solution is better in what situations? How can we tell which to use?

If all we were concerned with was making sure that the inferences about *hasPossession* will be supported, then there would be no reason to prefer one solution over the other. But modeling in the semantic web is not just about supporting desired inferences, but also about supporting reuse. Might someone else want to use this model, in a slightly different way? A future query is just as likely to be interested in *hasPossession* as *possessedBy*. Furthermore, we might in the future wish to combine *hasPossession* (or *possessedBy*) with another property. For this reason, one might choose to use both solutions together, by using *inverseOf* and *subPropertyOf* together in a systematic way. That is, by specifying inverses for every property, regardless of the *subPropertyOf* level. In this case, this results in

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

The systematicity of this structure can be more readily seen in figure 7-1.



7-1 Systematic combination of inverseOf and subPropertyOf

The most attentive reader might have one more concern about the systematicity of figure 7-1. In particular, the selection of which properties are the subject of *owl:inverseOf* and which are the object (in the diagram, which ones go on the left or on the right of the diagram) is arbitrary. Shouldn't there be three more *owl:inverseOf* triples, pointing from right to left? Indeed there should, but there is no need to assert these triples, as we shall see in the next challenge.

7.1.2 Challenge: Using the modeling language to extend the modeling language

It is not unusual for beginning modelers to look at the list of constructs defined in OWL, and to say, "There is a feature of the OWL language that I would like to use, that is very similar to the ones that are included. Why did they leave it out? I would prefer to build my model using a different set of primitives." In many cases, the extra language feature that they desire is actually already supported by OWL, as a combination of other features. It is a simple matter of using these features in combination.

CHALLENGE

RDFS allows me to specify that one class is a *subClassOf* another, but I prefer to think the other way around (perhaps because of the structure of some legacy data I want to work with), and specify that something is *superClassOf* something else. That is, I want the parent class to be the subject of all the definitional triples. Using my own namespace *myowl:* for this desired relation, I would like to have the triples look like this:

```
Food myowl:superClassOf BakedGood;
     myowl:superClassOf Confectionary;
     myowl:superClassOf PackagedFood;
     myowl:superClassOf PreparedFood;
     myowl:superClassOf ProcessedFood .
```

If we instead use *rdfs:subClassOf*, all the triples go the other way around; *Food* will be the object of each triple, and all the types of *Food* will be the subjects.

Since OWL does not provide a *superClassOf* resource (or to speak more correctly, OWL does not define any inference rules that will provide any semantics for a *superClassOf* resource), what can we do?

SOLUTION

What do we want *myowl:superClassOf* to mean? For every triple of the form

```
P myowl:superClassOf Q .
```

We want to be able to infer that

```
Q rdfs:subClassOf P .
```

This can be accomplished simply by declaring an inverse:

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

It is a simple application of the rule for *owl:inverseOf* to see that this accomplishes the desired effect. Nevertheless, this is not a solution that many beginning modelers think of. It seems to them that they have no right to modify or extend the meaning of the OWL language; that they cannot make statements about the OWL and RDFS resources (like *rdfs:subClassOf*). But remember the AAA slogan of RDF: Anyone can say Anything about Any Topic. In particular, a modeler can say things about the resources defined in the standard.

In fact, we can take this slogan so far as to allow a modeler to say

```
rdfs:superClassOf owl:inverseOf rdfs:subClassOf .
```

This differs from the previous triple in that the subject is a resource in the (standard) RDFS namespace. The RDF Slogan allows a modeler to say this, and indeed, there is nothing in the standards that will prevent it. However, referring to a resource in the RDFS namespace is likely to suggest to human readers of the model that this relationship is part of the RDFS standard. Since one purpose of a model is to communicate to other human beings, it is generally not a good idea to make statements that are likely to be misleading, so we do not endorse this practice.

Selecting namespaces for resources that extend the capabilities of the OWL language is a delicate matter; in the next chapter, we will examine a case study in which this has been done in a careful way.

7.1.3 Challenge: The marriage of Shakespeare

In section 4.5 XXX we lamented that even though we had asserted that Anne Hathaway had married Shakespeare, that we did not know that Shakespeare had married Anne Hathaway. We are now in a position to remedy that.

CHALLENGE

How can we infer marriages in the reverse direction from which they are asserted?

SOLUTION

Simply declare *bio:married* to be it own inverse, thus:

```
bio:married owl:inverseOf bio:married .
```

Now any triple that uses *bio:married* will automatically be inferred to hold in the other direction. In particular, now when we assert

```
bio:AnneHathaway bio:married lit:Shakespeare .
```

we can infer that

```
lit:Shakespeare bio:married bio:AnneHathaway .
```

This pattern of self-inverses is so common, that it has been built into OWL using a special construct called *owl:SymmetricProperty*.

7.2 Symmetric Properties

owl:inverseOf relates one property to another. The special case in which these two properties are the same (as was the case for *bio:married* for the Shakespeare example) is common enough that the OWL committee has provided a special name for it, *owl:SymmetricProperty*. Unlike *owl:inverseOf*, which is a property that relates two other properties, *owl:SymmetricProperty* is just an aspect of a single property, and is expressed in OWL as a Class. We express that a property is symmetric in the same way as we express membership in any class, i.e.,

```
P rdf:type owl:SymmetricProperty .
```

As usual, the meaning of this statement is given by the inferences that can be drawn from it. From this triple, we can infer that

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

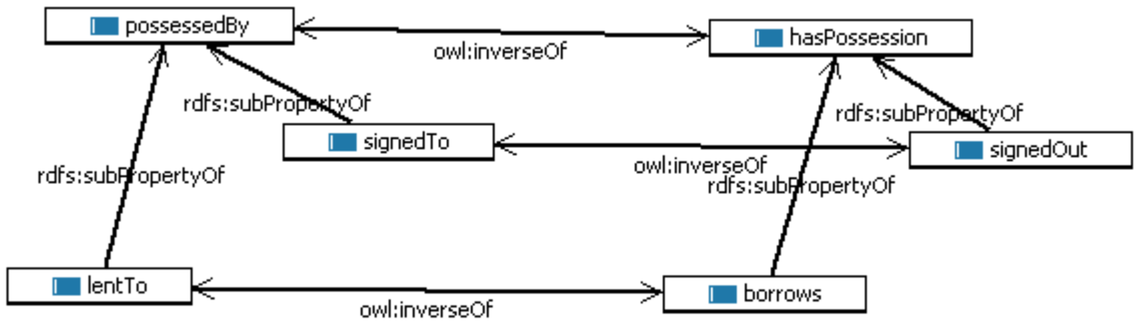
So, in the case of marriage of Shakespeare, we can assert

```
    bio:married rdf:type owl:SymmetricProperty .
```

7.2.1 Using OWL to extend OWL

As we describe more and more of the power of the OWL modeling language, there will be more and more opportunities to define at least some aspects of a new construct in terms of previously defined constructs. We can use this method to streamline our presentation of the OWL language.

We have seen a need for this already in figure 7-1, in which we noticed that we had expressed all of our inverses in one direction, while we observe that we really need to have them go both ways, as shown in figure 7-2.



7-2 Systematic combination of inverseOf and subPropertyOf; contrast figure 7-1, with one-directional inverses.

We asserted the triples from left to right, namely

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

We would like to be able to infer the triples from right to left, namely

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

CHALLENGE

How can we infer all of these triples without having to assert them?

SOLUTION

Since we want *owl:inverseOf* to work in both directions, this can be done easily by asserting that *owl:inverseOf* is its own inverse, thus:

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

You might have had to do a double-take to read that triple, that *owl:inverseOf* is its own inverse. Fortunately, we now have a more readable and somewhat more understandable way to say this, namely

```
owl:inverseOf rdf:type owl:SymmetricProperty .
```

In either case, we get the inferences we desire for figure 7-2, in which the inverses point both ways. This also means that all the inferences in section 7.1.1, in both directions, will always be found.

7.3 Transitivity

In mathematics, a relation *R* is said to be *transitive* if *R(a,b)* and *R(b,c)* implies *R(a,c)*. The same idea is used for the OWL construct *owl:TransitiveProperty*. Just like *owl:SymmetricProperty*, *owl:TransitiveProperty* is a class of properties, so a model can assert that a property is a member of the class

```
P rdf:type owl:TransitiveProperty .
```

The meaning of this is given by a somewhat more elaborate rule than we have seen so far in this chapter. Namely, if we have two triples of the form

```
X P Y .
Y P Z .
```

we can infer that

```
X P Z .
```

Notice that there is no need for even more elaborate rules like

```
A P B .
B P C .
C P D .
```

Implies

```
A P D .
```

since this conclusion can be reached by applying the simple rule over and over again.

Some typical examples of transitive properties include ancestor/descendant (if Victoria is an ancestor of Edward, and Edward is an ancestor of Elizabeth, then Victoria is an ancestor of Elizabeth), and containment (if Osaka is in Japan and Japan is in Asia, then Osaka is in Asia).

7.3.1 Challenge: Relating parents to ancestors

A model of genealogy will typically include notions of parents as well as ancestors, and we'd like them to fit together. But parents are not transitive (my parents' parents are not my parents), while ancestors are.

CHALLENGE

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

SOLUTION

Start by defining the parent property to be a *subPropertyOf* the ancestor property, thus:

```
:hasParent rdfs:subPropertyOf :hasAncestor .
```

Then declare ancestor only to be a transitive property:

```
:hasAncestor rdf:type owl:TransitiveProperty .
```

Let's see how this works on some examples.

```
:Alexia :hasParent :WillemAlexander .
:WillemAlexander :hasParent :Beatrix .
:Beatrix :hasAncestor :Wilhelmina .
```

Because of the *subPropertyOf* relation between *hasParent* and *hasAncestor*, and the fact that *hasAncestor* is a *TransitiveProperty*, we can infer that

```
Alexia hasAncestor WillemAlexander .
WillemAlexander hasAncestor Beatrix .
Alexia hasAncestor Beatrix .
WillemAlexander hasAncestor Wilhelmina .
Alexia hasAncestor Wilhelmina .
```

Information about the heritage is integrated, regardless of whether it originated with *hasParent* or *hasAncestor*. Information about *hasParent*, on the other hand, is only available as it was directly asserted, as it was not declared to be transitive. The results of this inference are shown in Figure 7-3.

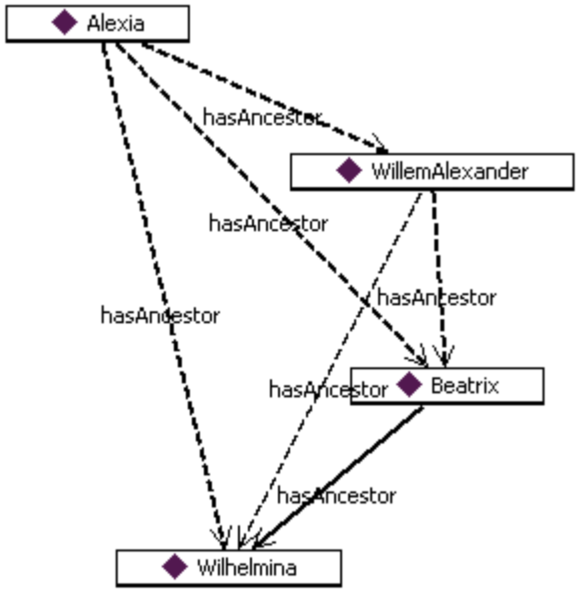


Figure 7-3. Inferences from transitive properties.

7.3.2 Challenge: Layers of relationships

Sometimes it can be somewhat controversial, whether a property is transitive or not. For instance, the relationship that is often expressed by the words “part of” in English is sometimes transitive (a piston is part of the engine, the engine is part of the car; is the piston part of the car?) and sometimes not (Mick Jagger’s thumb is part of Mick Jagger, and Mick Jagger is part of The Rolling Stones. Is Mick Jagger’s thumb part of The

Rolling Stones?). In the spirit of anticipating possible uses of a model, it is worthwhile to support both points of view, whenever there is any chance that controversy might arise.

CHALLENGE

Simultaneously maintain transitive and non-transitive versions of the *partOf* imformation.

SOLUTION

Define two versions of the *partOf* property, in different namespaces, with one a *subPropertyOf* the other, with the superproperty declared as transitive:

```
dm:partOf rdfs:subPropertyOf gm:partOf .
gm:partOf rdf:type owl:TransitiveProperty .
```

Depending on which interpretation of *partOf* any particular application needs, it can query the appropriate property. For those who prefer to think that Mick Jagger's thumb is not part of the Rolling Stones, the original *dm:partOf* property is useful. For those who instead consider that Mick Jagger's Thumb is a part of the Rolling Stones, the transitive superproperty *gm:partOf* is appropriate.

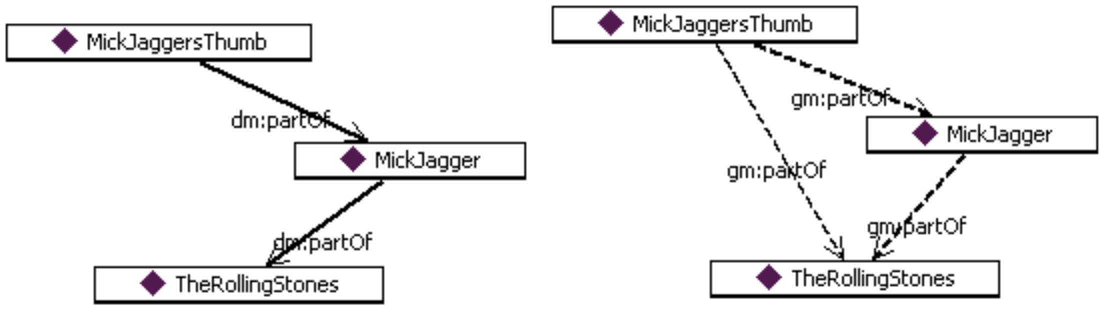


Figure 7-4. Different interpretations of partOf.

7.3.3 Managing networks of dependencies

The same modeling patterns we have been using to manage relationships (like ancestry) or set containment (like part of) can be used just as well in a very different setting, namely, to manage networks of dependencies. In the following series of challenges, we will see how the familiar constructs of *rdfs:subPropertyOf*, *owl:inverseOf* and *owl:TransitiveProperty* can be combined in novel ways to model important aspects of such networks.

A common application of this idea is in workflow management. In a complex working situation, a variety of tasks must be repeatedly performed in a set sequence. The idea of workflow management is that the sequence can be represented explicitly, and the progress of each task tracked in that sequence. Why would someone want to model workflow in a semantic web? For the same reason one wants to put anything on the web – so that parts of the workflow can be shared with others, encouraging reuse, review and publication of work fragments.

Real workflow specifications are far too detailed to serve as examples in a book, so we will use a simple example to show how it works. Let's make some ice cream, using the following recipe:

> Slice a vanilla bean lengthwise, and scrape the contents into 1 cup of heavy cream. Bring the mixture to a simmer, but do not boil.
>
> While the cream is heating, separate three eggs. Add ½ cup white sugar, and beat until fluffy. Gradually add the warm cream, beating constantly. Return the custard mixture to medium heat, and cook until mixture leaves a heavy coat on the back of a spatula. Chill well. Combine custard with 1 cup whole milk, and turn in ice cream freezer according to manufacturer's instructions.

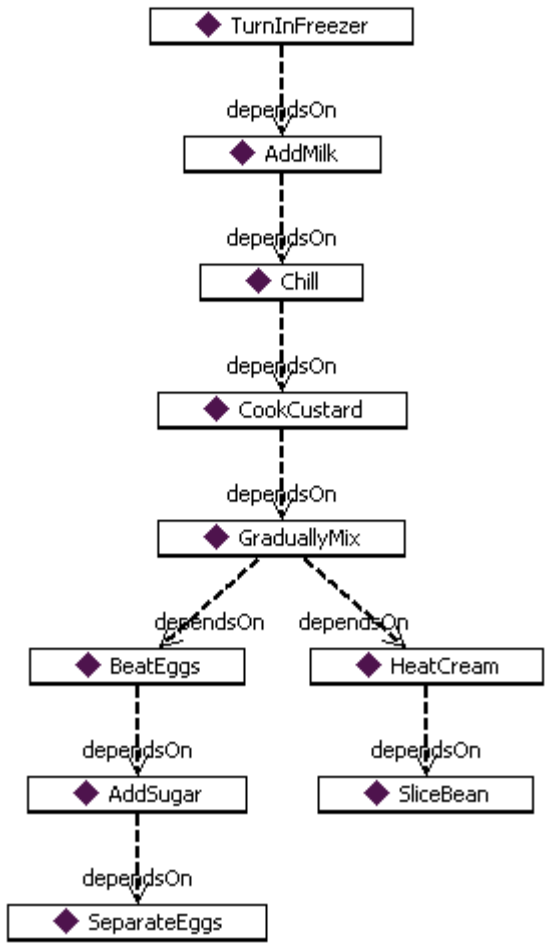
First, let's use a property *dependsOn* to represent the dependencies between the steps, and define its inverse *enables*, since each step enables the next in the correct execution of the workflow:

```
:dependsOn owl:inverseOf :enables .
```

Now we can define the dependency structure of the recipe steps:

```
:SliceBean :enables :HeatCream .
:SeparateEggs :enables :AddSugar .
:AddSugar :enables :BeatEggs
:BeatEggs :enables :GraduallyMix .
:HeatCream :enables :GraduallyMix .
:GraduallyMix :enables :CookCustard .
:CookCustard :enables :Chill .
:Chill :enables :AddMilk .
:AddMilk :enables :TurnInFreezer .
```

Because of the *inverseOf*, we can view these steps either in enabling order as asserted, or in dependency order, as show in figure 7-5.



7-5 Dependencies for homemade ice cream

CHALLENGE

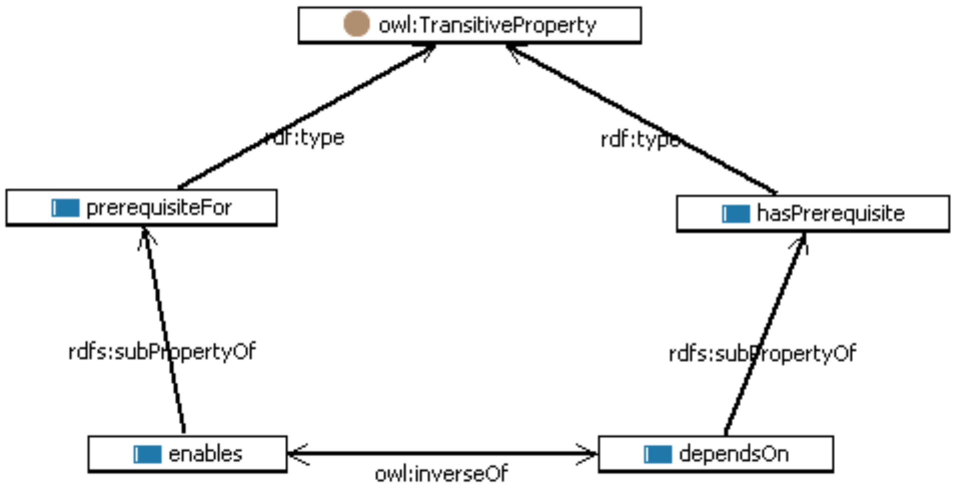
For any particular step in the process, we might want to know all the steps it depends on, or all the steps that depend on it. How can we do this, using the patterns we already know?

SOLUTION

We can use the *subPropertyOf/TransitiveProperty* pattern (as we used, e.g., in section 7.3.1) for each of *dependsOn* and *enables* as follows:

```
:dependsOn rdfs:subPropertyOf :hasPrerequisite .
:hasPrerequisite rdf:type owl:TransitiveProperty .
:enables rdfs:subPropertyOf :prerequisiteFor .
:prerequisiteFor rdf:type owl:TransitiveProperty .
```

These relationships can be seen graphically in 7-6



7-6. Transitive Properties hasPrerequisite and prerequisiteFor defined in terms of dependsOn and enables.

From these triples, for instance, we can infer that *GraduallyMix* has five prerequisites, namely

```
:GraduallyMix :hasPrerequisite :AddSugar ;
              :hasPrerequisite :SeparateEggs ;
              :hasPrerequisite :SliceBean ;
              :hasPrerequisite :HeatCream ;
              :hasPrerequisite :BeatEggs .
```

CHALLENGE

In a more realistic workflow management setting, we wouldn't just be managing a single process (corresponding to a single recipe). We would be managing several processes that interact in complex ways. We could even lose track of which steps are in the same procedure. Is there a way to find out, given a particular step, the other steps in the same process? In our recipe example, can we model the relationship between steps, so that we can connect steps in the same recipe together?

SOLUTION

First, we combine together both of our fundamental relationships (*enables* and *dependsOn*) as common *subPropertyOf* a single unifying property (*neighborStep*). We then in turn make that a *subPropertyOf* of a transitive property (*inSameRecipe*), shown here in N3 and as a diagram in figure 7-7(a).

```
:dependsOn rdfs:subPropertyOf :neighborStep .
:enables rdfs:subPropertyOf :neighborStep .
:neighborStep rdfs:subPropertyOf :inSameRecipe .
:inSameRecipe rdf:type owl:TransitiveProperty .
```

These triples can be seen graphically in figure 7-7(a).

What inferences can we draw from these triples, for the instance *GraduallyMix*? Any directly related step (related by either *dependsOn* or *enables*) becomes a *neighborStep*, and any combination of neighbors is rolled up with *inSameRecipe*. A few selected inferences are shown here:

```
:GraduallyMix :neighborStep :BeatEggs ;
              :neighborStep :HeatCream ;
              :neighborStep :CookCustard .
:CookCustard :neighborStep :Chill ;
              :neighborStep :GraduallyMix .
:GraduallyMix :inSameRecipe :BeatEggs ;
              :inSameRecipe :HeatCream ;
              :inSameRecipe :CookCustard .
:CookCustard :inSameRecipe :Chill ;
              :inSameRecipe :GraduallyMix .
…
:GraduallyMix :inSameRecipe :AddMilk ;
              :inSameRecipe :CookCustard ;
              :inSameRecipe :TurnInFreezer ;
              :inSameRecipe :AddSugar ;
              :inSameRecipe :SeparateEggs ;
              :inSameRecipe :SliceBean ;
              :inSameRecipe :HeatCream ;
              :inSameRecipe :GraduallyMix ;
              :inSameRecipe :Chill ;
              :inSameRecipe :BeatEggs .
```

All the steps in this recipse have been gathered up with *inSameRecipe*, as desired. In fact, any two steps in this recipe will be related to one another by *inSameRecipe*, including relating each step to itself. In particular, the triple

```
GraduallyMix inSameRecipe GraduallyMix .
```

has been inferred. While this is, strictly speaking, correct (after all, *GraduallyMix* is indeed in the same recipe as *GraduallyMix*), it might not be what we actually wanted to know.

CHALLENGE

How can we define a property that will relate a recipe step only to the *other* steps in the same recipe?

SOLUTION

Earlier we defined two properties, *hasPrerequisite* and *prerequisiteFor*, one looking "downstream" along the dependencies, and the other looking "upstream",

```
:dependsOn rdfs:subPropertyOf :hasPrerequisite .
:hasPrerequisite rdf:type owl:TransitiveProperty .
:enables rdfs:subPropertyOf :prerequisiteFor .
:prerequisiteFor rdf:type owl:TransitiveProperty .
```

If we join these two together under a common superproperty which is not transitive,

```
:hasPrerequisite rdfs:subPropertyOf :otherStep .
:prerequisiteFor rdfs:subPropertyOf :otherStep .
```

These relationships are shown diagrammatically in figure 7-7(b)

We track the inferences separately for each property. For *hasPrerequisite*, we have already seen that we can infer

```
:GraduallyMix :hasPrerequisite :AddSugar   ;
              :hasPrerequisite :SeparateEggs   ;
              :hasPrerequisite :SliceBean   ;
              :hasPrerequisite :HeatCream   ;
              :hasPrerequisite :BeatEggs  .
```

For *prerequisisteOf*, we get the following inferences:

```
:GraduallyMix :prerequisiteFor :AddMilk ;
              :prerequisiteFor :CookCustard ;
              :prerequisiteFor :TurnInFreezer ;
              :prerequisiteFor :Chill .
```

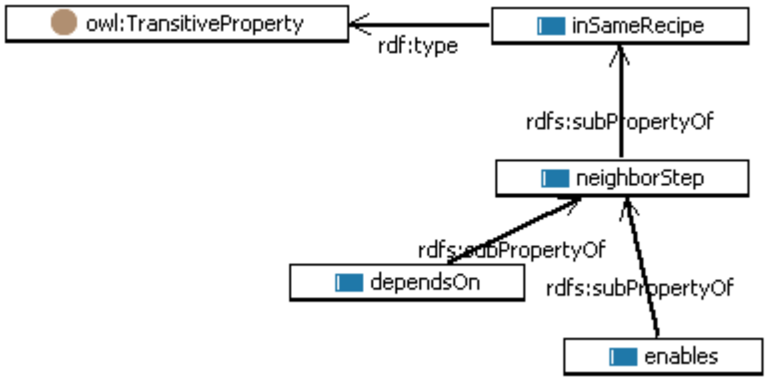
Now, for *otherStep*, we get the combination of these two. Notice that neither list includes *GraduallyMix* itself, so it does not appear in this list, either:

```
:GraduallyMix :otherStep :AddMilk ;
              :otherStep :CookCustard ;
              :otherStep :TurnInFreezer ;
              :otherStep :AddSugar ;
              :otherStep :SeparateEggs ;
              :otherStep :SliceBean ;
              :otherStep :HeatCream ;
              :otherStep :Chill ;
              :otherStep :BeatEggs .
```

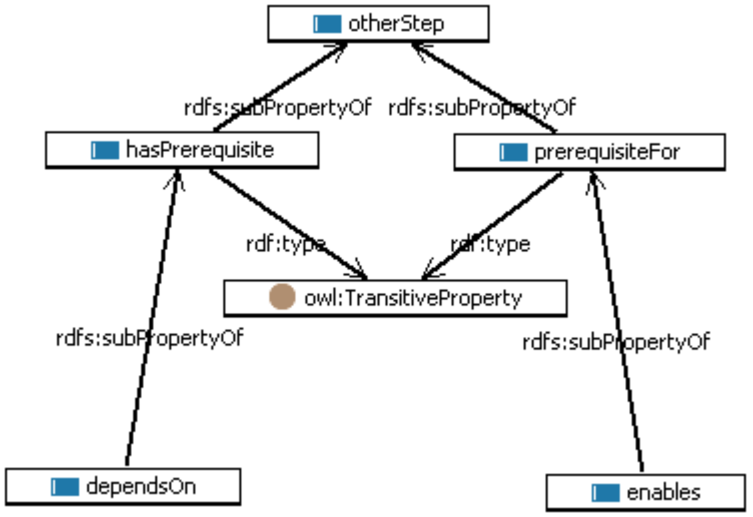
Figure 7-7 shows the two patterns; for *inSameRecipe*, we have a single transitive property at the top of a *subPropertyOf* tree; both primitive properties (*enables* and *dependsOn*) are brought together, and any combination of the resulting property (*neighborStep*) are chained together as a *TransitiveProperty* (*inSameRecipe*). For *otherStep*, the top-level property itself is *not* transitive, but is a simple combination (via two *subPropertyOf* links) of two transitive properties (*hasPrerequisite* and *prerequisiteFor*). Inference for each of these transitive properties is done separately from the other, and the results combined (without any more transitive interaction). Hence, for *inSameRecipe*, the reflexive triples like

```
    :GraduallyMix :inSameRecipe :GraduallyMix .
```

are included, while for *otherStep*, they are not.



(a)



(b)

7-7 Contrast patterns for inSameRecipe (includes self) and otherStep (excludes self). Both patterns work from the same input properties *dependsOn* and *enables*, but yield different results.

7.4 Equivalence

RDF provides a global notion of identity that has validity across data sources; that global identity is the URI. This makes it possible to refer to a single entity in a distributed way. But when we want to merge information from multiple sources controlled by multiple stakeholders, it is unlikely that any two stakeholders will use the same URI to refer to the same entity. Thus, in a federated information setting, it is useful to be able to stipulate that two URIs actually refer to the same entity. But there are different ways in which two entities can be the same – some are more equal than others. OWL-Fast provides a variety of notions of equivalence. As with other constructs in OWL, these different constructs are defined by the inferences they entail.

7.4.1 Equivalent Classes

In Section 6.3.4 XXX, we used a simple idiom to express that one class had the same elements as another; in particular, we asserted two triples

```
:Analyst rdfs:subClassOf :Researcher .
:Researcher rdfs:subClassOf :Analyst .
```

to indicate that every *Analyst* is a *Researcher*, and that every *Researcher* is an *Analyst*. As we saw in section 6.3.4XXX, the rule for *rdfs:subClassOf* can be applied in each direction, to support the necessary inferences to make every *Analyst* a *Researcher* and vice versa. When two classes are known to always have the same members, we say that the classes are equivalent. The pattern shown above allows us to express class equivalence in RDFS, if in a somewhat unintuitive way.

OWL-Fast provides a more intuitive expression of class equivalence, using the construct *owl:equivalentClass*. A single triple expresses class equivalence in the obvious way:

```
:Analyst owl:equivalentClass :Researcher .
```

As with any other construct in RDFS or OWL, the precise meaning of *owl:equivalentClass* is given by the inferences that can be drawn. In particular, if we know that

```
A owl:equivalentClass  B .
r rdf:type A .
```

then we can infer

```
r rdf:type B .
```

So far, this is just the type propagation rule that we used to define the meaning of *rdfs:subClassOf* back in chapter 6. But owl:equivalentClass has another rule as well; that

```
A rdfs:subClassOf  B .
r rdf:type B .
```

then we can infer

```
r rdf:type A .
```

That is, the two classes A and B have exactly the same members.

It seems a bit of a shame that something as simple as equivalence requires two rules to express, especially when the rules are so similar. In fact, this isn't necessary; if we observe that

```
owl:equivalentClass rdfs:type owl:SymmetricProperty .
```

then there is no need for the second rule; we can infer it from the first rule, and the symmetry of equivalentClass.

In fact, we don't actually need any rules at all; if we also assert that

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

we can use the rules for *subPropertyOf* and *subClassOf* to infer everything about *equivalentClass*! Let's see how the rules for OWL that we have already learned work for *owl:equivalentClass*, in the case of the *Analyst* and the *Researcher*.

From the rule about *rdfs:subClassOf* and the statement of equivalence of *Analyst* and *Researcher*, we can infer

```
:Analyst rdfs:subClassOf :Researcher .
```

But since *owl:equivalentClass* is symmetric, we can also infer that

```
:Researcher owl:equivalentClass :Analyst .
```

and by applying the rule for *rdfs:subClassOf* once again, we get

```
:Researcher rdfs:subClassOf :Analyst .
```

That is, simply by applying what we already know about *rdfs:subClassOf* and *owl:SymmetricProperty*, we can infer both *rdfs:subClassOf* triples from the single *owl:equivalentClass* triple.

Notice that when two classes are equivalent, it only means that the two classes have the same members. Other properties of the classes are not shared; for example, each class keeps its own *rdfs:label*. This means that if these classes have been merged from two different applications, that each of these applications will still display the class by the original print name; only the members of the class will change.

7.4.2 Equivalent Properties

In section 6.3.6.1XXX, we saw how we could use *rdfs:subPropertyOf* to make two properties behave in the same way; the trick we used there was very similar to the double-subClassOf trick we used in section 6.3.4; we use *rdfs:subPropertyOf* twice to indicate that two properties are equivalent.

```
:borrows rdfs:subPropertyOf :checkedOut .
:checkedOut rdfs:subPropertyOf :borrows .
```

OWL-Fast also provides a more intuitive way to express property equivalence, using *owl:equivalentProperty*, as follows:

```
:borrows owl:equivalentProperty :checkedOut .
```

When two properties are equivalent, we expect that in any triple that uses one as a predicate, the other can be substituted, that is, for any triple

```
A borrows B .
```

we can infer

```
A checkedOut B .
```

And vice-versa. We can accomplish this in a manner analogous to the method used for *owl:equivalentClass* in section 7.4.1; we define *owl:equivalentProperty* in terms of other OWL-Fast constructs.

```
owl:equivalentProperty rdfs:subPropertyOf rdfs:subPropertyOf.
owl:equivalentProperty rdf:type owl:SymmetricProperty .
```

Starting with the asserted equivalence of *borrows* and *checkedOut*, using these triples, and the rules for *rdfs:subPropertyOf* and *owl:SymmetricProperty*, we can infer that

```
borrows rdfs:subPropertyOf checkedOut .
checkedOut owl:equivalentProperty borrows .
checkedOut rdfs:subPropertyOf borrows .
```

Once we have inferred that *borrows* and *checkedOut* are *rdfs:subPropertyOf* one another, we can make all the inferences mentioned in section 6.3.6.1XXX.

When we express new constructs (like *owl:equivalentProperty* in this section) to constructs we already know (*rdfs:subPropertyOf* and *owl:SymmetricProperty*), we explicitly describe how the various parts of the language fit together. That is, rather than just noticing that the rule that governs *owl:equivalentProperty* is the same rule as the one that governs *rdfs:subPropertyOf* (except that it works both ways!), we can actually model these facts; by making *owl:equivalentProperty* a subproperty of *rdfs:subPropertyOf*, we explicitly assert that they are governed by the same rule. By making *owl:equivalentProperty* a *SymmetricProperty*, we assert the fact 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.

7.4.3 Same individuals

Class equivalence (*owl:equivalentClass*) and property equivalence (*own:equivalentProperty*) provide intuitive ways to express relationships that were already expressible in RDFS; in this sense, neither of these constructs have increased the expressive power of OWL-Fast beyond what was already available in RDFS; they have just made it easier to express and clearer to read. These constructs refer respectively to classes of things and the properties that relate them

But when we are describing things in the world, we aren't only describing classes and properties; we are describing the things themselves. These are the members of the classes. We refer to these as *individuals*. We have encountered a number of individuals in our examples so far; Wenger the Analyst, Twelfth Night the Play, Shakespeare the Playwright, Kildare the Surgeon, Kaneda the All Star Player, and any number of things whose class membership has not been specified; Wales, The Firm, and Moby Dick. But remember the non-unique naming assumption from section 1.7.4; often our information comes from multiple sources that might not have done any coordination in their reference to individuals. How do we handle the situation in which we determine that two individuals that we originally thought of separately are in fact the same individual?

In OWL-Fast, this is done with the single construct *owl:sameAs*. Our old friend William Shakespeare will provide us with an example of how *owl:sameAs* works.

From Chapter 3 XXX, we have the following triples about the literary career of William Shakespeare:

```
lit:Shakespeare lit:wrote lit:AsYouLikeIt  ;
                lit:wrote lit:HenryV ;
                lit:wrote lit:LovesLaboursLost ;
                lit:wrote lit:MeasureForMeasure ;
                lit:wrote lit:TwelfthNight ;
                lit:wrote lit:WintersTale ;
                lit:wrote lit:Hamlet ;
                lit:wrote lit:Othello .
```

Suppose we have at our disposal information from the Stratford Parish Register, which lists the following information from some baptisms that occurred there. We will use *spr:* as the namespace identifier for URIs from the Stratford Parish Register.

```
spr:Gulielmus spr:hasFather spr:JohannesShakspere .
spr:Susanna spr:hasFather spr:WilliamShakspere .
spr:Hamnet spr:hasFather spr:WilliamShakspere .
spr:Judeth spr:hasFather spr:WilliamShakspere .
```

Suppose that our research determines that indeed, the resources mentioned here as *spr:Gulielmus*, *spr:WilliamShakespere* and *lit:Shakespeare* all refer to the same individual, so that the answer to the question, "Did Hamnet's father write Hamlet?" would be "yes." If we had known that all of these things refer to the same person in advance of having represented the Stratford Parish Register in RDF, we could have used the same URI (e.g., lit:Shakespeare) for each occurrence of the Bard. But now it is too late; the URIs from each data source have already been chosen. What is to be done?

First, let's think about how to pose the question "Did Hamnet's father write Hamlet?" We can write this as a graph pattern in SPARQL, as follows:

```
{spr:Hamnet spr:hasFather ?d .
 ?d lit:wrote lit:Hamlet . }
```

that is, we are looking for a single resource that links Hamnet to Hamlet via the two links *spr:hasFather* and *lit:wrote*.

In OWL-Fast, we have the option of asserting the sameness of two resources. Let's start with just one

```
spr:WilliamShakspere owl:sameAs lit:Shakespeare .
```

The meaning of this triple, as always in OWL-Fast, is expressed by the inferences that can be drawn. The rule for *owl:sameAs* is quite intuitive; it says that if A *owl:sameAs* B, then in any triple where we see A, we can infer the same triple, with A replaced by B. So for our *Shakespeare* example, we have that for any triple of the form

```
spr:WilliamShakespere P O .
```

we can infer

```
lit:Shakespeare P O .
```

Similarly, for any triple of the form

```
S P spr:WilliamShakespeare .
```

we can infer

```
S P lit:Shakespeare .
```

Furthermore, just as we did for *owl:equivalentClass* (section 7.4.1) and *owl:equivalentProperty* (section 7.4.2), we assert that *owl:sameAs* is a *owl:SymmetricProperty*:

```
owl:sameAs rdf:type owl:SymmetricProperty .
```

This allows us to infer that

```
lit:Shakespeare owl:sameAs spr:WilliamShakspere.
```

so that we can replace any occurrence of *lit:Shakespeare* with *spr:WilliamShakspere* as well.

Let's see how this works with the triples we know from literary history and the Register. We list all triples; asserted triples in plain face, inferred triples in *italics*. Among the inferred triples, we begin by replacing *lit:Shakespeare* with *spr:WilliamShakspere*, then continue by replacing *spr:WilliamShakspere* with *lit:Shakespeare*:

```
lit:Shakespeare lit:wrote lit:AsYouLikeIt  ;
                lit:wrote lit:HenryV ;
                lit:wrote lit:LovesLaboursLost ;
                lit:wrote lit:MeasureForMeasure ;
                lit:wrote lit:TwelfthNight ;
                lit:wrote lit:WintersTale ;
                lit:wrote lit:Hamlet ;
                lit:wrote lit:Othello .
spr:Gulielmus spr:hasFather spr:JohannesShakspere .
spr:Susanna spr:hasFather spr:WilliamShakspere .
spr:Hamnet spr:hasFather spr:WilliamShakspere .
spr:Judeth spr:hasFather spr:WilliamShakspere .
spr:WilliamShakspere
                lit:wrote lit:AsYouLikeIt  ;
                lit:wrote lit:HenryV ;
                lit:wrote lit:LovesLaboursLost ;
                lit:wrote lit:MeasureForMeasure ;
                lit:wrote lit:TwelfthNight ;
                lit:wrote lit:WintersTale ;
                lit:wrote lit:Hamlet ;
                lit:wrote lit:Othello .
spr:Gulielmus spr:hasFather spr:JohannesShakspere .
spr:Susanna spr:hasFather lit:Shakespeare .
spr:Hamnet spr:hasFather lit:Shakespeare .
spr:Judeth spr:hasFather lit:Shakespeare .
```

Now the answer to the query, “Did Hamnet’s father write Hamlet?” is “yes,” since there is a binding for the variable *?d* in the SPARQL graph pattern given above. In fact, there are two possible bindings, *?d=lit:Shakespeare* and *?d=spr:Shakspere*.

7.4.3.1 Challenge: Merging data from different databases

In section 3.3 XXX, we saw how to interpret information in a table as RDF triples. Each row in the table became a single individual, and each cell in the table became a triple. The subject of the triple is the individual corresponding to the row that the cell is in; the predicate is made up from the table name and the field name, and the object is the cell contents. The table in table 3-10XXX (repeated here as Table 7-1) becomes 63 triples, for the 7 fields and 9 rows.

Product						
ID	**Model No.**	**Division**	**Product Line**	**Manufacture location**	**SKU**	**Available**
1	ZX-3	Manufacturing support	Paper machine	Sacramento	FB3524	23
2	ZX-3P	Manufacturing support	Paper machine	Sacramento	KD5243	4
3	ZX-3S	Manufacturing support	Paper machine	Sacramento	IL4028	34
4	B-1430	Control Engineering	Feedback Line	Elizabeth	KS4520	23
5	B-1430X	Control Engineering	Feedback Line	Elizabeth	CL5934	14
6	B-1431	Control Engineering	Active Sensor	Seoul	KK3945	0
7	DBB-12	Accessories	Monitor	Hong Kong	ND5520	100
8	SP-1234	Safety	Safety Valve	Cleveland	HI4554	4
9	SPX-1234	Safety	Safety Valve	Cleveland	OP5333	14

Table 7-1. Sample tabular data for triples, from table 3-10.

Let's look at just the triples having to do with the Manufacture_Location.

```
man:Product1 man:Product_Manufacture_Location Sacramento .
man:Product2 man:Product_Manufacture_Location Sacramento .
man:Product3 man:Product_Manufacture_Location Sacramento .
man:Product4 man:Product_Manufacture_Location Elizabeth .
man:Product5 man:Product_Manufacture_Location Elizabeth .
man:Product6 man:Product_Manufacture_Location Seoul .
man:Product7 man:Product_Manufacture_Location Hong Kong .
man:Product8 man:Product_Manufacture_Location Cleveland .
man:Product9 man:Product_Manufacture_Location Cleveland .
```

Suppose that another division in the company keeps its own table of the products, with information that is useful for that division's business activities; namely, it describes the sort of facility that is required to produce the part. Table 7-2 shows some products and the facilities they require.

Product		
ID	**Model No.**	**Facility**
1	B-1430	Assembly Center
2	B-1431	Assembly Center
3	M13-P	Assembly Center
4	ZX-3S	Assembly Center
5	ZX-3	Factory
6	TC-43	Factory
7	B-1430X	Machine Shop
8	SP-1234	Machine Shop
9	1180-M	Machine Shop

Table 7-2 Sample data; parts and the facilities required to produce them.

Some of the products in Table 7-2 also appeared in Table 7-1, others did not. It is not uncommon for different databases to overlap in such an inexact way.

CHALLENGE

Using the products that appear in both tables, how can we write a federated query that will cross-reference cities with the facilities that are required for the production that takes place there?

SOLUTION

If these two tables had been in a single database, then there could have been a foreign-key reference from one table to the other, and we could have joined the two tables together. But since the tables come from two different databases, there is no such common reference.

When we turn both tables into triples, the individuals corresponding to each row are assigned global identifiers. Suppose that we use the namespace *p:* for this second database; when we turn Table 7-2 into triples, we get 27 triples, for the nine rows and three fields. The triples corresponding to the required facilities are:

```
p:Product1     p:Product_Facility  "Assembly Center" .
p:Product2     p:Product_Facility  "Assembly Center" .
p:Product3     p:Product_Facility  "Assembly Center" .
p:Product4     p:Product_Facility  "Assembly Center" .
p:Product5     p:Product_Facility  "Factory" .
p:Product6     p:Product_Facility  "Factory" .
p:Product7     p:Product_Facility  "Machine Shop" .
p:Product8     p:Product_Facility  "Machine Shop" .
p:Product9     p:Product_Facility  "Machine Shop" .
```

Although we have global identifiers for individuals in these tables, those identifiers are not the same. For instance, *p:Product1* is the same as *man:Product4* (both correspond to model number B-1430). How can we cross-reference from one table to the other?

The answer is to use a series of *owl:sameAs* triples, as follows:

```
p:Product1 owl:sameAs man:Product4 .
p:Product2 owl:sameAs man:Product6 .
p:Product4 owl:sameAs man:Product3 .
p:Product5 owl:sameAs man:Product1 .
p:Product7 owl:sameAs man:Product5 .
p:Product8 owl:sameAs man:Product8 .
```

Now if we match the following SPARQL graph pattern

```
{?p p:Product_Facility ?facility .
 ?p man:Product_Manufacture_Location ?location .}
```

And display *?facility* and *?location* in Table 7-3, we get

?location	?facility
Elizabeth	Assembly Center
Seoul	Asssembly Center
Sacramento	Assembly Center

Sacramento	Factory
Elizabeth	Machine Shop
Cleveland	Machine Shop

Table 7-3 Locations cross-referenced with facilities, computed via products.

This solution has addressed the challenge for the particular data in the example, but the solution relied on the fact that we knew which product from one table matched with which product from another table. But *owl:sameAs* only solves part of the problem; in real data situations, in which the data in the tables changes frequently, it is not practical to assert all the *owl:sameAs* triples by hand. In the next section, we will see how OWL-Fast provides a solution to the rest of the challenge.

7.5 Computing sameness – functional properties

Functional Properties in OWL get their name from a concept in mathematics, but like most of the OWL constructs, they have a natural interpretation in everyday life. A function property is one for which there can be just one value. Examples of such properties are quite common; *hasMother* (since a person has just one biological mother), *hasBirthplace* (someone was born in just one place) and *birthdate* (just one) are a few simple examples.

In mathematics, a *function* is a mapping that gives one value for any particular input; so x^2 is a function, since for any value of x, there is exactly one value for x^2. Another way to say this is that if $x=y$, then $x^{2=}y^2$. In order to solve the previous challenge problem, we will have to have constructs in OWL-Fast that have this same sort of behavior; that is, we want to describe something as being able to refer to only a single value.

The next two constructs we will describe, *FunctionaProperty* and *InverseFunctionalProperty,* use this idea to determine when two resources refer to the same individual, thereby providing the OWL modeler with a means for describing how information from multiple sources are to be considered as a distributed web of information. The constructs provide the semantic framework for using OWL-Fast in a Semantic Web setting.

7.5.1 Functional properties

OWL-Fast borrows the name *functional* to describe a property that, like a mathematical function, can only take one value for any particular individual. The precise details of the meaning of *owl:FunctionalProperty* is given, as usual, as an inference pattern. If we have the following triples

```
P rdf:type owl:FunctionalProperty .
X P A .
X P B .
```

Then we can infer that

```
A owl:sameAs B .
```

This definition of *owl:FunctionalProperty* is analogous to the mathematical situation in which we know that x^2 has a single unambiguous value. More precisely, if we know that $x^2 = a$ and $x^2 = b$, then we may conclude that $a=b$. In OWL-Fast, this looks as follows, in which the first three triples are asserted, and the fourth is inferred:

```
math:hasSquare rdf:type owl:FunctionalProperty .
x math:hasSquare A .
x math:hasSquare B .
A owl:sameAs B .
```

Functional properties are important in OWL-Fast because they allow sameness to be inferred. For instance, suppose that in the Stratford Parish Registry we have an entry that tells us

```
lit:Shakespeare fam:hasFather bio:JohannesShakspere .
```

and that from Shakespeare's grave we learn that

```
lit:Shakespeare fam:hasFather bio:JohnShakespeare .
```

We would like to conclude that John and Johannes are in fact the same person. If we know from a background model of family relationships that

```
fam:hasFather rdf:type owl:FunctionalProperty .
```

then we can conclude, from the definition of *owl:FunctionalProperty*, that

```
bio:JohannesShakspere owl:sameAs bio:JohnShakespeare .
```

as desired.

While *owl:FunctionalProperty* provides us with a means of concluding that two resources are the same, this is not the usual pattern for determining that two entities are the same in most real data. Much more common is the closely related notion of *owl:InverseFunctionalProperty*, which we treat next.

7.5.2 Inverse Functional Properties

Some people consider *owl:InverseFunctionalProperty* to be the most important modeling construct in OWL-Fast, especially in situations in which a model is being used to manage data from multiple sources. Whether this is true or not, it is certainly true that it has the most difficult name with respect to its utility of any construct.

The name *owl:InverseFunctionalProperty* was chosen to be consistent with the closely related *owl:FunctionalProperty*, and in fact, one can think of an *owl:InverseFunctionalProperty* simply as the inverse of an *owl:FunctionalProperty*. So, if *math:hasSquare* is a functional property, then its inverse, *math:hasSquareRoot,* is an inverse functional property.

What does this mean, in terms of inferences that can be drawn? The rule looks very similar to the rule for *owl:FunctionalProperty*. If we know that

```
P rdf:type owl:InverseFunctionalProperty .
A P X .
B P X .
```

then we can infer that

```
A owl:sameAs B .
```

For example, if we define a property *buriedAt* to be sufficiently specific that we cannot have two people buriedAt the same location, then we can declare it to be and *owl:InverseFunctionalProperty*. If we were then to have two triples that assert

```
spr:Shakespere buriedAt TrinityChancel .
lit:Shakespeare buriedAt TrinityChancel .
```

Then we could infer that

```
spr:Shakespere owl:sameAs lit:Shakespeare .
```

An *owl:InverseFunctionalProperty* plays the role of a key field in a relational database; a single value of the property cannot be shared by two entities, just as a key field may not be duplicated in more than one row. Unlike the case of a relational database, OWL-Fast does not signal an error if two entities are found to share a value for an inverse functional property; instead, OWL-Fast infers that the two entities must be the same. Because of the non-unique naming assumption, we cannot tell that two entities are distinct just by looking at their URIs.

Examples of inverse functional properties are fairly commonplace; any identifying number (social security number, employee number, driver's license number, serial number, etc.) is an inverse functional property. In some cases, full names are inverse functional properties, though in most applications, name duplications (is it the same John

Smith?) are common enough that full names are not inverse functional properties. In an application at the Boston Children's Hospital, it was necessary to find an inverse functional property that would uniquely identify a baby (since newborns don't always have their social security numbers assigned yet). The added catch was that it had to be a property that the mother was certain, or at least extremely likely, to remember. Although babies are born at any time of day in a busy hospital, it is sufficiently unusual for two babies to born at exactly the same minute that time of birth could be used as an inverse functional property. And every mother was able to remember when her baby was born.

Now that we have inverse functional properties, we are able to continue the solution to the challenge from Section 7.4.3.1. In that solution, we merged information from two databases by matching the global URIs of individuals from two databases with the following series of owl:sameAs triples

```
p:Product1 owl:sameAs man:Product4 .
p:Product2 owl:sameAs man:Product6 .
p:Product4 owl:sameAs man:Product3 .
p:Product5 owl:sameAs man:Product1 .
p:Product7 owl:sameAs man:Product5 .
p:Product8 owl:sameAs man:Product8 .
```

Once we had these triples, we were able to cross-reference cities with facilities, using products as an intermediary. But we had to create these triples by hand.

CHALLENGE

How can we infer the appropriate *owl:sameAs* triples, from the data that has already been asserted?

SOLUTION

The approach we will take to this challenge is to find an inverse functional property that is present in both data sets, that we can use to bridge between them. When we examine Table 7-2 and in table 3-10XXX, we see that they both have a field called *Model No***.**, which refers to the identifying model number of the product. As is typical for such identifying numbers, if two products have the same model number, then they are the same product. So we want to declare Model No. to be an inverse functional property, thus:

```
man:Product_ModelNo rdf:type owl:InverseFunctionalProperty .
```

This almost works, but there is still a catch; each database has its own Model No. property; the one in this triple came from the database in Chapter 3XXX; in this chapter, there is another property. *p:Product_ModelNo*. So it seems that we still have more integration to do. Fortunately, we already have the tool we need to do this; we simply have to assert that these two properties are equivalent, thus:

```
p:Product_ModelNo owl:equivalentProperty man:Product_ModelNo .
```

It really doesn't matter which way around we do any of these things; since *owl:equivalentProperty* is symmetric, we can write this triple with the subject and object reversed, and it will make no difference to the inferences.

Let's see how these inferences roll out. We begin with the asserted triples from both data sources, and proceed with inferred triples:

```
p:Product1 p:Product_ModelNo "B-1430" .
p:Product2 p:Product_ModelNo "B-1431" .
p:Product3 p:Product_ModelNo "M13-P" .
p:Product4 p:Product_ModelNo "ZX-3S" .
p:Product5 p:Product_ModelNo "ZX-3" .
p:Product6 p:Product_ModelNo "TC-43" .
p:Product7 p:Product_ModelNo "B-1430X" .
p:Product8 p:Product_ModelNo "SP-1234" .
p:Product9 p:Product_ModelNo "1180-M" .
man:Product1 man:Product_ModelNo "ZX-3" .
man:Product2 man:Product_ModelNo "ZX-3P" .
man:Product3 man:Product_ModelNo "ZX-3S" .
man:Product4 man:Product_ModelNo "B-1430" .
man:Product5 man:Product_ModelNo "B-1430X" .
man:Product6 man:Product_ModelNo "B-1431" .
man:Product7 man:Product_ModelNo "DBB-12" .
man:Product8 man:Product_ModelNo "SP-1234" .
man:Product9 man:Product_ModelNo "SPX-1234" .
p:Product1 man:Product_ModelNo "B-1430" .
p:Product2 man:Product_ModelNo "B-1431" .
p:Product3 man:Product_ModelNo "M13-P" .
p:Product4 man:Product_ModelNo "ZX-3S" .
p:Product5 man:Product_ModelNo "ZX-3" .
p:Product6 man:Product_ModelNo "TC-43" .
p:Product7 man:Product_ModelNo "B-1430X" .
p:Product8 man:Product_ModelNo "SP-1234" .
p:Product9 man:Product_ModelNo "1180-M" .
p:Product1 owl:sameAs man:Product4 .
p:Product2 owl:sameAs man:Product6 .
p:Product4 owl:sameAs man:Product3 .
p:Product5 owl:sameAs man:Product1 .
p:Product7 owl:sameAs man:Product5 .
p:Product8 owl:sameAs man:Product8 .
```

The last six triples are exactly the *owl:sameAs* triples that we needed to complete our challenge.

While this use *of owl:InverseFunctionalProperty* works fine for an example like this, most real data integration situations rely on more elaborate notions of identity, that include multiple properties as well as uncertainty (what about that one freak day when two babies were born the same minute at the same hospital?). Addressing these issues is the subject of current development on the OWL standard.

7.5.3 Combining Functional and Inverse Functional Properties.

It is possible and often very useful for a single property to be both an *owl:FunctionalProperty* and an *owl:InverseFunctionalProperty*. When a property is in both of these classes, then it is effectively a *one-to-one* property; that is, for any one individual, there is exactly one value for the property, and vice versa. In the case of identification numbers, it is usually desirable that the property be one-to-one, as the following challenge illustrates:

CHALLENGE

Suppose we want to assign identification numbers to students at a university. These numbers will be used to assign results of classes (grades) as well as billing information for the students. Clearly no two students should share an identification number, and neither should one student be allowed to have more than one identification number. How do we model this situation in OWL-Fast?

SOLUTION

Define a property *hasIdentityNo* that associates a number with each student, so that its domain and range are defined by

```
hasIdentityNo rdfs:domain Student .
hasIdentityNo rdfs:range Integer .
```

Furthermore, we can enforce the uniqueness properties by asserting that

```
hasIdentityNo rdf:type owl:FunctionalProperty .
hasIdentityNo rdf:type owl:InverseFunctionalProperty .
```

Any two students who share an identity number must be the same (since it is Inverse Functional); furthermore, each student can have at most one identity number (since it is Functional).

7.5.4 Functional and Inverse Functional Summary

It is meaningful and useful to use Functional and Inverse Functional in isolation or together; for example:

Functional only: *hasMother* is a functional property only. Someone has exactly one mother, but many people can share the same mother.

Inverse Functional Only: *hasDiary* is an inverse functional property only. A person may have many diaries, but it is the nature of a diary that it is not a collaborative effort; it is authored by one person only.

Both Functional and Inverse Functional: *taxID* is both inverse functional and functional, since we want there to be exactly one *taxID* for each person, and vice versa.

7.6 A few more constructs

OWL-Fast provides a small extension to the vocabulary beyond RDFS, but these extensions greatly increase the scope of applicability of the language. In the preceding examples, we have seen how these new features interact with the features of RDFS to provide a richer modeling environment. The inclusion of *owl:inverseOf* combines with *rdfs:subClassO*f by allowing us to align properties that might not have been expressed in compatible ways in existing data schemas. The inclusion of *owl:TransitiveProperty* combines with *rdfs:subPropertyOf* in a number of novel combinations, as seen here, allowing us to model a variety of relationships among chains of individuals. The most applicable extensions, from a Web perspective, are those that deal with sameness of different individuals. *sameAs*, *FunctionaProperty*, and *InverseFunctionalProperty* in particular provide the OWL modeler with a means for describing how information from multiple sources are to be considered as a distributed web of information. The constructs provide the semantic framework for using OWL-Fast in a Semantic Web setting.

OWL provides a few more distinctions which, while they do not provide any semantics to a model, provide some useful discipline and provide information that many editing tools can take advantage of when displaying models. When displaying what value some property takes for some subject, should the display be a link to another object, or a widget for a particular data type? Tools that get this right seem intuitive and easy to use; tools that don't seem awkward. So OWL provides a way to describe properties that can help a tool sort this out. In OWL, this is done by distinguishing between *owl:DatatypeProperty* and *owl:ObjectProperty*.

In RDF, a triple always has a resource as its subject and predicate, but can have either another resource as object, or it can have a data item of some XML data type. We have seen plentiful examples of both of these:

```
ship:QEII ship:maidenVoyage "May 2, 1969" .
man:Product1 man:Product_SKU "FB3524" .
AnneHathaway bio:married lit:Shakespeare .
GraduallyMix inSameRecipe BeatEggs .
spr:Susanna spr:hasFather spr:WilliamShakspere .
```

Most tools that deal with OWL at this time prefer to make the distinction; in this case, *ship:maidenVoyage* and *man:Product_SKU* are datatype properties, while *bio:married*, *inSameRecipe* and *spr:hasFather* are object properties. In triples, we say:

```
ship:maidenVoyage rdf:type owl:DatatypeProperty .
man:Product_SKU rdf:type  owl:DatatypeProperty .
bio:married rdf:type owl:ObjectProperty .
inSameRecipe rdf:type owl:ObjectProperty .
spr:hasFather rdf:type owl:ObjectProperty .
```

Another distinction that is made in OWL (for reasons which are mostly historical; in future versions of OWL, we expect this distinction to be removed) is the difference between *rdfs:Class* and *owl:Class*.

In Chapter 6XXX, we introduced the notion of *rdfs:Class* as the means by which schema information could be represented in RDF. Since that time, we have introduced a wide array of "schema-like" constructs like inverse, subproperty, transitivity, etc. But OWL also provides a special case of *rdfs:Class* called *owl:Class*. Since OWL is based on RDFS, it was an easy matter to make *owl:Class* backward compatibly with *rdfs:Class* by saying that every member of *owl:Class* is also a member of *rdfs:Class*. This statement needn't be made in prose, since we can say it in RDFS. In particular, the OWL specification stipulates that

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

Most tools today insist that classes used in OWL models be declared as members of *owl:Class*. In this chapter, we have left these class declarations out, since they are not central to the modeling examples we were giving. But implicit in the examples in this chapter, as such statements as

```
Food  rdfs:type owl:Class .
BakedGood rdfs:type owl:Class .
Confectionary rdfs:type owl:Class .
PackagedFood rdfs:type owl:Class .
PreparedFood rdfs:type owl:Class .
ProcessedFood  rdfs:type owl:Class .
man:Product rdfs:type owl:Class .
p:Product rdfs:type owl:Class .
```

Most OWL tools today will work more consistently if classes are defined as instances of *owl:Class*; most model editors will do this automatically when a class is created.

7.7 OWL-Fast: Summary

The constructs in OWL-Fast are a subset of the constructs in OWL, but this subset provides considerable flexibility for modeling in the Semantic Web. In the next chapter, we will see some examples of how OWL-Fast is used in large scale Semantic Web projects. Here we summarize the constructs in OWL-Fast for easy reference, along with a (very) brief gloss of how the constructs are interpreted by an inference engine.

RDF Schema Features:

rdfs:subClassOf: Members of subclass are also member of superclass

rdfs:subPropertyOf: Relations described by subproperty also hold for superproperty

rdfs:domain: The subject of a triple is classified into the domain of the predicate.

rdfs:range The object of a triple is classified into the range of the predicate

Annotation Properties:

rdfs:label: No inferential semantics, printable name

rdfs:comment: No inferential semantics, information for readers of the model.

OWL Features - Equality:

equivalentClass: Members of each class are also members of the other.

equivalentProperty: Relations that hold for each property also hold for the other

sameAs: All statements about one instance hold for the other**.**

OWL Features: Property Characteristics:

inverseOf: Exchange subject and object.

TransitiveProperty: Chains of relationships collapse into a single relationship.

SymmetricProperty: A property that is its own inverse.

FunctionalProperty: Only one value allowed (as object).

InverseFunctionalProperty: Only one value allowed (as subject)

ObjectProperty: Property can have resource as object.

DatatypeProperty: Property can have data value as object.