DRAFT: Please do not distribute without permission. Thanks!

A New Axiomatic Semantics for Semantic Web Languages

Christopher Menzel and (eventually!) Patrick Hayes

Introduction

In [McG/Fikes], McGinness and Fikes provided a commendable "axomatic semantics" of the Semantic Web languages RDF, RDF Schema, and DAML+OIL. The idea was to express each language as a first-order logical theory whose axioms captured the intended semantics of each term in the language's reserved vocabulary. Useful as this document has been, two significant developments have dated it:

- DAML+OIL has been supplanted by the various OWLs – OWL Lite, OWL DL, and OWL Full; and
- The W3C has published detailed and precise model theories for each of the languages.

Consequently, the time is ripe for an update. A good bit of this task has been done in [Hayes-SW2CL], and I have drawn freely from that work here. However, Hayes's focus there is on translation, which masks the pure axiomatic semantics somewhat. He also makes free use of Common Logic sequence variables, which are avoided here in anticipation of the use of these axioms with first-order reasoners, and makes greater use of lists and their properties in his axioms. And overall, the current approach differs from Hayes in its approach to the axiomatization of OWL.

In this document, then, for each of the languages RDF, RDF Schema, and OWL I will first lay out its basic syntax – its basic lexicon and reserved vocabulary, and its grammar – and its basic formal model theory, comporting as closely as possible with the syntactic and semantic specifications of these languages provided by the appropriate W3C documents. Then I will provide a mapping of each language into the Common Logic CL framework (http://cl.tamu.edu); specifically, I will "embed" the intended semantics of each language into CL in the form of a logical theory, i.e., a set of axioms whose interpretations should (provably) correspond in a meaning-preserving way to the intended interpretations of the source languages relative to their semantics.

As a follow-on to this project, I will describe a general architecture for integration involving the construction of "bridge axioms" that connect the concepts expressed in different ontologies. I will also examine strategies for automating the discovery of plausible bridge axioms. Finally, as adumbrated above, I will endeavor to code up a toy implementation of the approach using an appropriate classical theorem prover such as Otter (http://www-unix.mcs.anl.gov/AR/otter) or Tau (http://hsinfosystems.com/taujay).

RDF Syntax

RDF Vocabularies

- A set of URI references, or "URIrefs"[1]
  - *ref*$_1$, *ref*$_2$, …
    - Logically speaking, URI refs are just *individual constants*
  - Reserved Vocabulary (http://www.w3.org/1999/02/22-rdf-syntax-ns)
    - Classification Vocabulary
      - Class refs: Property, XMLLiteral
      - Property ref: type
    - Container Vocabulary
      - Class ref: Bag, Seq, Alt
      - Property refs: _1, _2, …
    - Collection Vocabulary
      - Class ref: List
      - Individual ref: nil
      - Property refs: first, rest
        - Use `rdf:parseType="Collection"` in RDF/XML
    - Reification Vocabulary
      - Class ref: Statement
      - Property refs: subject, predicate, object
    - "Idiomatic" Vocabulary (for use with structured properties)
      - Property ref: value
- Blank nodes _:x, _:y, _:z, …
- Literals
  - *Plain*
    - "abc", where "abc" is a finite string of appropriate characters
  - *Typed*
    - "abc"^^ *ref*

A *term* is a uriref, blank node, or a literal.

Say that one vocabulary V2 *extends* another V1 just in case V1 ⊆ V2.

A mapping from a set B of blank nodes into a vocabulary V is said to be a *grounding of B in V*.

Grammar: RDF Triples and RDF Graphs

An *RDF-triple* is (abstractly speaking) a 3-tuple ⟨S,P,O⟩, where P is a URIref, S is either a URIref or a blank node, and O is any term. A triple is *ground* if it contains no blank nodes and *plain* if it contains no literals.

---

[1] See http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/#dfn-URI-reference.

I'll follow Hayes in and let "ex:" stand for an abbreviation of a full URL (ending in a pound sign) for some arbitrary namespace, and for terms in known namespaces like those for RDF we'll let assume the relevant prefix and also omit outside angle brackets,[2] e.g.,

- <ex:Cain> rdf:type <ex:Adam>
- <http://www.w3.org/TR/2004/REC-rdf-mt-20040210> dc:Creator <http://www.ihmc.us/users/phayes>

An *RDF graph* is a set of RDF triples. A graph is *ground* (*plain*) if all of its triples are ground (plain).

Let B be the blank nodes of an RDF graph G and let *g* be a grounding of B in a vocabulary V* that extends the vocabulary of G. Then the *grounding* o*f G by g*, *g*[G], is the result of replacing every blank node *b* in every triple of G with *g*(*b*).

A Basic Semantics for RDF Grammar

All of the W3C ontology languages are based upon the same basic grammar defined initially for RDF; we will therefore refer to all of these languages as *RDF languages*. Our approach in this document will be to provide a basic "structural" model theory for the basic RDF grammar common to the W3C ontology languages; we will refrain from interpreting the reserved vocabulary of any of the languages. Rather, we will express the intended semantics of the reserved vocabulary in each case (to the extent that this is possible) axiomatically, as (first-order) *theories*. This will give us a uniform measure of expressive power: The power of an ontology language will be the set of theorems it entails or, equivalently,[3] the set of basic models that it has. One language L2 will be more powerful than another L1, with regard to a certain set of shared vocabulary V, the theorems of L1 that expressed in terms of V alone are a proper subset of the theorems of L2 that are similarly expressed.

Our basic model theory for RDF grammar is based on Hayes' [RDF-Semantics] but is somewhat simpler in its treatment of blank nodes.) Building on ideas from the semantics of Common Logic (see [Hayes/Menzel]), the RDF universe is largely "homogeneous". There is a basic division between value spaces and resources (in fact, the former are assumed to be defined independently), but the domain of resources within any (formal) interpretation of a vocabulary will include "individuals", classes, and properties alike without making any logical significant discrimination; everything in it is a logical "first-class citizen". The trick for pulling this off is to assign an extension – a set, or set of

---

[2] Specifically, we assume the following:

- @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
- @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
- @prefix owl: <http://www.w3.org/2002/07/owl#> .

[3] Given the equivalence between theoremhood and entailment guaranteed by the completeness theorem for first-order languages.

pairs, of resources to every resource – the properties of the domain will be those resources whose extension consists of an ordered pair whose first-element is a resource and whose second element is a resource (i.e., an individual property value) or a data value. We assume there is a fixed stock B of blank nodes.

An *interpretation* I for an RDF language, then, is a 4-tuple $\langle IR,IEXT,IS,V\rangle$ consisting of a vocabulary V; a nonempty set IR of resources that includes a distinguished subset IP; an extension mapping $IEXT : IR \rightarrow Pow(IR) \cup Pow(IR^2)$, and a mapping IS of $V \cup B$ into IR. (We assume also that such that $B \cap IR$ is empty, though doesn't really matter.)[4]

A plain, ground RDF triple $\langle S,P,O\rangle$ is *true* in I just in case $\langle IS(S),IS(O)\rangle \in IEXT(IS(P))$, i.e., just in case the resource or value indicated by O is a property value assigned to the resource indicated by S by the property indicated by P. A plain ground graph G is true just in case all of its component triples are true.

Let G be an arbitrary plain graph. Then G is true in I just in case there is an interpretation $I^* = \langle IR,IEXT,IS^*,V\rangle$ such that IS* differs from IS at most only on what it assigns to the blank nodes of G such that G is true in I*. (The semantics of literals is omitted for simplicity and will be added later.)

Mapping RDF into FOL and CL

As noted, the above is a model theory for the basic RDF grammar alone. It builds in no intrinsic semantics for any of the distinguished URIrefs of RDF or RDFS. From the point of view of first-order logic, the set of graphs generated by the grammar of RDF for a given vocabulary corresponds to a class of relatively weak sublanguages of the class of first-order languages. Specifically, given an RDF vocabulary V, the language L(V) is a first-order language containing

1. Conjunction and the existential quantifier
2. The class B of blank nodes as its individual variables
3. The vocabulary V as its individual constants

---

[4] One might think that, as befits their intended role as properties, we should stipulate that for $p \in IP$, $IEXT(p) \in Pow(IR^2)$. But in fact RDFs flexibility is such as to allow a resource to play any role in an RDF triple, and this feature nicely reflects the fact that one and the same resource might, from one perspective, be a property, and from another an individual or a class. For instance, suppose that '`ex:studentID`' indicates a property of students. But it might also be useful to consider this *also* to indicate the *class* of student IDs. By allowing for the possibility that resources can include both individual resources as well as pairs of resources in the extension of a given resource, we could let '`ex:studentID`' both property and class roles by imposing the condition on its denotation *ID* that $\langle s,i\rangle \in IEXT(ID)$ only if $i \in IEXT(ID)$. Note that, in [RDF Semantics], Hayes captures this flexibility by introducing a separate class extension function taking resources to sets of resources when he extends the RDF model theory to RDFS, which is of course equivalent to the approach here.

4. A distinguished 3-place predicate `HasProp`.

The translation *trans* of an RDF triple ⟨*Sub*,*Prop*,*Obj*⟩ generated from elements of V into this language is straightforward: *trans*(⟨S,P,O⟩) is simply `(HasProp` *Prop Sub Obj*`)`. (We are using the ASCII-based first-order language KIF here.) The translation of an RDF graph {T_1, …, T_n}, then, is simply

`(exists (`var_1 … var_m`) (and` *trans*(T_1) … *trans*(T_n))

where var_1 … var_m correspond to all the blank nodes occurring in G. (See [McGuinness and Fikes, "An Axiomatic Semantics for RDF, RDF-S, and DAML+OIL".)

An interpretation I = ⟨IR,IEXT,IS,V⟩ of an RDF vocabulary V can be transformed easily into a standard first-order interpretation M(I) of L(V). The *domain* of M is IR, and IS is the interpretation of the individual constants and variables of L(V) (i.e., the URIrefs of V and the blank nodes). The extension of `HasProp` is simply $\bigcup${IEXT(r) : $r \in$ IP}, i.e., the union of the extensions assigned to the set IP of "properties" in IR by IEXT. It is nearly trivial to show that a graph G is true in an interpretation I of V if and only if its translation *trans*(G) is true in its first-order transformation M(I).

Because of its greater structural affinity with both RDF syntax and semantics, however, it is preferable to use the Common Logic (CL) framework rather than a more traditional first-order framework for translating RDF – see http://cl.tamu.edu. In a CL dialect – we will use the KIF-like dialect of CL known as "CLIF" – just as there are no restriction on where a URIref can occur in a triple, so there are no restrictions on where a CL name can occur; indeed, there is no distinction in CL between names, predicates, and function symbols; there are only names that can play all three syntactic roles within an atomic sentence. (Names even play the role of variables in quantification.) Because of the far greater syntactic freedom of CL, the translation from RDF into an existential/conjunctive version of CLIF – call it EC-CLIF) is even more straightforward. Specifically, the translation *CLIFtrans* of an RDF triple ⟨S,P,O⟩ generated from elements of V into CLIF is simply (P S O). And, as above the translation of an RDF graph {T_1, …, T_n} is

`(exists (`var_1 … var_m`) (and` *CLIFtrans*(T_1) … *CLIFtrans*(T_n)).

Additionally, the notion of an interpretation for EC-CLIF is *identical* to that of an RDF interpretation; no transformations are necessary, so it is trivial that an RDF graph is true in an interpretation I iff its CLIF translation is true in I.

Two approaches to RDF Mapping

As noted, our model theory for RDF only interprets the basic grammatical structures of RDF: triples and graphs. It does not attempt to fix the semantics of RDF's reserved vocabulary. Were we to be considering RDF as an end in itself, it would of course be wise to do so. However, our goal here is an architecture for integration, and a fundamental piece of that architecture will involve the translation of Web languages into (a dialect of) CL.

From this perspective, the model theory above is completely adequate: It fixes the interpretation of the basic RDF grammatical structures adequately enough for us to be able to determine the correctness of various translation schemes from RDF into a first-order framework. Within this far more expressive framework, then, we can fix the meanings of the *reserved* RDF vocabulary by mapping RDF sentences that use them in a meaning preserving into CLIF, and use the CLIF images of those sentences to reason upon RDF ontologies.

As Hayes points out in http://www.ihmc.us/users/phayes/CL/SW2SCL.html, there are two ways to think about mapping RDF (or Semantic Web languages generally) into CL: *translation* and *embedding*. The translation of an RDF sentence S simply captures in CL the intended meaning of S while making no pretense of preserving the reserved vocabulary of RDF. For instance, the intended meaning of 'rdf:type' is essentially just the membership relation between an individual and a class, which, in turn, in standard FOL, can be expressed straightaway in terms of atomic predication. Hence a natural *translation* of the RDF graph:

{ihmc:Hayes rdf:type CyCL:Person, ihmc:employerOf rdf:type rdf:Property}

would be simply the sentences:

```
(CyCL:Person ihmc:Hayes)

(rdf:Property ihmc:employerOf).
```

Embedding, by contrast, preserves a structural copy of the RDF triples in a given source language graph (as many as necessary, at any rate), thereby retaining the reserved vocabulary of the source language, which is axiomatized directly. Thus, an embedding strategy for the 2-element RDF graph above first preserves the triples:

```
(RDF-Triple ihmc:Hayes rdf:type CyCL:Person)

(RDF-Triple ihmc:employerOf  rdf:type  rdf:Property).
```

"`RDF-Triple`" and "`rdf:type`" are then axiomatized directly:

```
(forall (P S O)
        (if (RDF-Triple S P O) (P S O))

(forall (x P)
        (if (rdf:type x P)
            (P x)
```

from which it is now possible to prove the translation above.

We note that the axiomatization of the two terms above involves the use of universal quantification and the conditional, which obviously take us beyond the inherent expressive powers of RDF. (The same will be true of the translation strategy as well as soon as one provides semantic axioms for lexical primitives like "`rdf:list`" that,

unlike "`rdf:type`", cannot be represented in terms of the bare machinery of first-order logic.)

However, our goal here is not simply to develop an alternative, first-order syntax for RDF. Rather, it is our intention to capture the *semantics* of the W3C languages, that is the intended meanings of the lexical primitives of the language as reflected in the model theories of those languages. Thus, from the beginning we will be embedding the languages into full first-order logic in the form of first-order theories; what we will add as we move first from RDF to RDFS, and from RDFS to OWL are additional lexical primitives and corresponding first-order axioms for those primitives.

Embedding RDF in FOL: RDF as a Logical Theory

We will adopt the embedding strategy in our approach we believe it will be important always to have a structural copy of the ontologies that one is integrating; it should also facility more accurate mappings into and out of CL before and after any automated reasoning takes place. The two axioms just given – for '`RDF-Triple`' and '`rdf:type`' – comprise the most fundamental axioms of RDF; again:

```
(forall (P S O)
        (if (RDF-Triple S P O) (P S O))

(forall (x P)
        (if (rdf:type x P)
            (P x)
```

Note that we don't want to use biconditionals here. For we don't want to assume – at the outset, anyway – that every instance of binary predication involving any property and any two objects is necessarily a state of affairs that RDF could recognize in the form of an RDF triple; similarly, we don't want to assume that every unary predication involving any object and any class indicates an rdf:type classification. In particular, there might be objects that cannot, for one reason or another, be considered legitimate RDF resources. (We might, for example, be considering an RDF ontology within the context of a larger universe.)

Supplementing the axioms just given, we also have that rdf:type is itself of the type rdf:property:

```
(RDF-Triple rdf:type rdf:type rdf:Property),
```

from which it follows from the previous axioms simply that

```
(rdf:Property rdf:type).
```

Indeed, we can say, more generally, of the predicate of any triple:

```
(forall (P S O)
        (if (RDF-Triple S P O)
            (rdf:Property P)))
```

Container Theory

There are so few semantic constraints on RDF containers and the so-called "_N-properties" that one can scarcely axiomatize them. Intuitively, containers (at least, seqs and bags; the status of "alts" is unclear) are sets of one ilk or another; specifically, seqs are ordered sets and bags (formally) are indexed sets, where the indices themselves have no intrinsic order. The first thing we can do is provide a set of axioms that connect the _N-properties to containers. Since there is no theoretical upper bound on the numbers *n* for which there is an associated _N-property, we can only present the axioms completely in the form of a schema; so for any numeral *NUM* we have the axiom:

```
(forall (C)
        (if (exists (x) (rdf:_NUM C x))
            (or (rdf:Bag C) (rdf:Seq C) (rdf:Alt))))
```
[5]

To these we can also add a schema classifying the N-properties as such:

```
(rdf:Property rdf:_NUM)
```

RDF List Theory

We begin with axioms that classify the basic vocabulary.

```
(rdf:Property rdf:first)

(rdf:Property rdf:rest)
```

We now axiomatize the list vocabulary. Note that the "intended" semantics of RDF rather dramatically exceeds what can actually be expressed in RDF. For example, there is no way in RDF proper to express that rdf:first and rdf:rest are functional properties; it is consistent with RDF to assume, for example, that a list could have two "first" elements. This raises an issue: on the one hand, we could axiomatize the "intended" semantics of RDF, and in particular flesh out the logic of RDF lists. On the other hand, we can let our logical theory reflect the *actual* expressive limitations of RDF and try therefore to let the theory be an accurate reflection of what is formally expressible in RDF.

We will choose the latter approach here. For one of the purposes of this paper is to compare and contrast the *formal* expressive capacities of each of the W3C ontology languages, the capacities that will be reflected in actual ontologies expressed in these various languages. A useful measure of relative strength, therefore, will be to identify interesting classes of sentences that one language entails – e.g., about lists – that another does not. Indeed, RDF, in particular, seems to give us only the following:[6]

---

[5] I should note that I have no clear idea at all of what the formal semantics of the Alt container is supposed to be.

[6] According to [RDF Schema, §5.2.4] it is implicit in the semantics that a triple of the form "L rdf:rest rdf:nil" expresses that L is a 1-element list. Arguably, however, this triple is true when L is rdf:nil;

```
; rdf:nil is an rdf:List
(rdf:List rdf:nil)
```

For simplicity, axioms for the reification vocabulary and the utility vocabulary will be omitted for the time being, as these concepts are not particularly robust, logically speaking. Axioms for literals also require delicate treatment and will be ignored for the time being.

RDFS Syntax

The grammar of RDFS is exactly that of RDF (as well as the various OWLs); it (and the OWLS) simply add to the reserved vocabulary. One can *refer* to things, data, datatypes, classes, and properties alike in RDF, but one cannot really *describe* classes and properties; there is in particular no way to group classes and properties themselves into "higher-order" classes, or indicate when one is a subclass or subproperty of another; or indicate when a given class is the domain or range of a given property. RDFS adds several additional primitives to RDF that makes this sort of explicit "schema" definition possible (http://www.w3.org/2000/01/rdf-schema). For simplicity

- RDFS Reserved Vocabulary
    - All reserved vocabulary of RDF
    - Classification Vocabulary
        - Class refs: Resource, Class, Literal, Container, Datatype, ContainerMembershipProperty
        - Property refs: domain, range, subClassOf, subPropertyOf, member
    - Utility Vocabulary
        - Property refs: comment, label, seeAlso, isDefinedBy

Embedding RDFS in FOL

We now provide axioms that capture the intended semantics of the new vocabulary introduced by RDFS. One of the things the addition of new vocabulary brings is the ability to express the intended semantics less expressive languages. Note we provide no axioms for the RDFS Utility Vocabulary, as the terms therein have no formal semantics.

---

moreover, the purported principle is not valid according to [RDF Semantics], neither is it confirmed by [RDF Primer], so we will not include an axiomatization of the purported principle here.

RDFS Logical Axioms

The broadest notion in RDF is that of a *resource*, which intuitively consists of everything that can be described in RDF; as [RDF Schema] succinctly puts it (Section 2.1): "This is the class of everything." Thus:

```
; Everything is an rdfs:Resource
(forall (x) (rdfs:Resource x))
```

Continuing the idea above, however, we will not assume that in every predication `(C x)` or `(P x y)` we might want to assert that `C` is an rdfs:Class or that `P` is an rdf:Property.

```
; rdfs:subClassOf axiom
(forall (c1 c2)
        (if (rdfs:subClassOf c1 c2)
            (forall (x)
                    (if (c1 x) (c2 x)))))

; Transitivity of rdfs:subClassOf
(forall (c1 c2)
        (if (and (rdfs:subClassOf c1 c2)
                 (rdfs:subClassOf c2 c3))
            (rdfs:subClassOf c1 c3)))


; Reflexivity of rdfs:subClassOf
(forall (c)
        (if (rdfs:Class c)
            (rdfs:subClassOf c c)))

; rdfs:subPropertyOf axioms
(forall (p1 p2)
        (if (and (rdf:Property p1) (rdfs:Property p2))
            (iff (rdfs:subPropertyOf p1 p2)
                 (forall (x y)
                         (if (p1 x y) (p2 x y))))))

; A property is a subproperty of itself
 (forall (p)
        (if (rdf:Property p)
            (rdfs:subPropertyOf p p)))

; domain axiom
 (forall (p c)
         (if (rdfs:domain p c)
             (forall (x)
                     (iff (exists (y) (p x y))
                          (c x)))))

; range axiom
(forall (p c)
        (if (rdfs:range p c)
            (forall (y)
                    (if (exists (x) (p x y))
```

```
                          (c y)))))

; Container axioms
(subclass rdf:Bag rdfs:Container)
(subclass rdf:Seq rdfs:Container)
(subclass rdf:Alt rdfs:Container)
(rdfs:subClassOf rdfs:ContainerMembershipProperty rdf:Property)
(rdfs:ContainerMembershipProperty rdf:_1)
(rdfs:ContainerMembershipProperty rdf:_2)
...
```

RDFS Classification and Typing Axioms

The introduction of rdfs:Class together with a number of more specific notions enables us to formulate a large collection of classification axioms.

```
; Classification Axioms: CLASSES
(rdfs:Class rdf:Property)
(rdfs:Class rdfs:Class)
(rdfs:Class rdfs:Resource)
(rdfs:Class rdf:Statement)
(rdfs:Class rdf:List)
(rdfs:Class rdf:Alt)
(rdfs:Class rdf:Bag)
(rdfs:Class rdf:Seq)
(rdfs:Class rdfs:Container)
(rdfs:Class rdfs:ContainerMembershipProperty)
(rdfs:Class rdfs:Literal)
(rdfs:Class rdfs:Datatype)

; Classification Axioms: PROPERTIES
(rdf:Property rdfs:domain)
(rdf:Property rdfs:range)
(rdf:Property rdfs:subPropertyOf)
(rdf:Property rdfs:subClassOf)
(rdf:Property rdfs:member)
```

The introduction of the properties rdfs:domain and rdfs:range enable us to introduce "typing" axioms for RDF and RDFS properties, i.e., axioms that constrain the kinds of the things on which given properties can be defined, and the kinds of values they can have on those things.

```
; Typing Axioms: DOMAINS
(rdfs:domain rdf:type rdfs:Resource)
(rdfs:domain rdfs:domain rdf:Property)
(rdfs:domain rdfs:range rdf:Property)
(rdfs:domain rdfs:subClassOf rdfs:Class)
(rdfs:domain rdfs:subPropertyOf rdf:Property)
(rdfs:domain rdf:subject rdf:Statement)
(rdfs:domain rdf:predicate rdf:Statement)
(rdfs:domain rdf:object rdf:Statement)
(rdfs:domain rdfs:member rdfs:Resource)
(rdfs:domain rdf:first rdf:List)
(rdfs:domain rdf:rest rdf:List)
```

```
(rdfs:domain rdf:_1 rdf:Container)
(rdfs:domain rdf:_2 rdf:Container)
...

; Typing Axioms: RANGES

(rdfs:range rdf:type rdfs:Class)
(rdfs:range rdfs:domain rdfs:Class)
(rdfs:range rdfs:range rdfs:Class)
(rdfs:range rdfs:subPropertyOf rdf:Property)
(rdfs:range rdfs:subClassOf rdfs:Class)
(rdfs:range rdf:subject rdfs:Resource)
(rdfs:range rdf:predicate rdfs:Resource)
(rdfs:range rdf:object rdfs:Resource)
(rdfs:range rdfs:member rdfs:Resource)
(rdfs:range rdf:first rdfs:Resource)
(rdfs:range rdf:rest rdf:List)
(rdfs:range rdf:_1 rdf:Resource)
(rdfs:range rdf:_2 rdf:Resource)
...
```

A Note on RDFS List Theory

RDFS adds nothing to the thin gruel provided by RDF for rdf:List other than thee general constraints above. Given the substantial use that RDF and RDFS make of the collection vocabulary to express facts about lists, the lack of of any formal semantic constraings might seem to be a rather glaring omission. Admittedly, requiring lists in RDF domains would make the semantics considerably more complex (and, depending on how it is done, no longer first-order). But it is hard to see how else is reasoning about lists in RDF(S) ontologies to be properly constrained. As seen below, the semantics of lists is shored up considerably in OWL.

OWL

As far as syntax and semantics and concerned, there is only one OWL. The phrases "OWL Lite", "OWL DL", and "OWL Full" are best thought of, not as noun phrases, but as adjectival phrases qualifying *ontologies* – an ontology is OWL Lite/DL/Full as the case may be depending on whether or not it satisfies certain conditions. These will be made explicit below.

OWL Reserved Vocabulary

- All reserved vocabulary of RDFS (hence RDF)
- OWL Reserved Vocabulary
    - Classification Vocabulary
        - Class refs: AllDifferent, Class, DataRange, DatatypeProperty, FunctionalProperty, InverseFunctionalProperty, Nothing,

ObjectProperty, Restriction, SymmetricProperty, Thing, TransitiveProperty

- Individual and class property refs: distinctMembers, differentFrom, disjointWith, equivalentClass, equivalentProperty, inverseOf, sameAs
- Class constructors: complementOf, intersectionOf, oneOf, unionOf
- Restriction property refs: allValuesFrom, cardinality, hasValue, maxCardinality, minCardinality, onProperty, someValuesFrom

OWL as a Logical Theory

OWL List Theory

Lists are fundamental to all of the Semantic Web languages, but it is only with OWL that lists are fully fleshed out. We note first that they are extensional, and that rdf:first and rdf:rest are functional properties – functionality is axiomatized below:

```
(forall (L1 L2)
        (if (and (rdf:List L1) (rdf:List L2))
            (iff (= L1 L2)
                 (forall (x L)
                         (and (iff (rdf:first L1 x)
                                   (rdf:first L2 x))
                              (iff (rdf:rest L1 L)
                                   (rdf:rest L2 L)))))))

(owl:FunctionalProperty rdf:first)

(owl:FunctionalProperty rdf:rest)
```

It is useful to introduce a membership property for lists; we use the list axioms found in [McG/Fikes]. Following the convention in [Hayes-SW2CL], we will write defined predicates in upper case.

```
(not (MEMBER x rdf:nil))

(forall (L1 L2 x)
        (if (and (rdf:List L1)
                 (rdf:List L2)
                 (owl:Thing x))
            (iff (MEMBER x L1)
                 (or (rdf:first L1 x)
                     (and (rdf:rest L1 L2)
                          (MEMBER x L2))))))
```

Only first in OWL are we guaranteed the existence of all finite lists of things and finite lists of classes.

```
(forall (x L1)
        (if (and (rdf:List L1)
                 (or (and (owl:Thing x)
                          (forall (y)
                                  (if (MEMBER y L1) (owl:Thing y))))
                     (and (owl:Class x)
                          (forall (y)
                                  (if (MEMBER y L1) (owl:Class y))))))
             (exists (L2)
                     (and (rdf:List L2)
                          (rdf:first L2 x)
                          (rdf:rest L2 L1)))))
```

Additionally, a list must consist entirely of owl:Things or entirely of owl:Classes:

```
(forall (L)
        (if (rdf:List L)
            (or (forall (x)
                        (if (MEMBER x L) (owl:Thing x)))
                (forall (x)
                        (if (MEMBER x L) (owl:Class x))))))
```

For purposes below, it will be useful also to introduce the NOREPEATSLIST predicate, as found in [McG/Fikes], more or less axiomatized here as there:

```
(forall (x L1)
        (if (and (rdf:first L1 x)
                 (rdf:rest L1 L2)
            (iff (NOREPEATSLIST L1)
                 (or (= L1 rdf:nil)
                     (and (not (MEMBER x L2)
                          (NOREPEATSLIST L2)))))))
```

Identity and Difference

```
(forall (x y)
        (if (and (owl:Thing x) (owl:Thing y))
            (iff (owl:sameAs x y) (= x y))))

(forall (x y)
        (if (and (owl:Thing x) (owl:Thing y))
            (iff (owl:differentFrom x y) (not (= x y)))))
```

owl:AllDifferent and owl:distinctMembers

Certain of OWL's constructs reflect the database world and other environments where domains tend to be finite and every object has a name. In such environments it can make sense to pick out classes all of whose members can be uniquely named. The OWL related constructs "owl:AllDifferent" and "owl:distinctMembers" reflect these conceptual roots. Intuitively, the members of "owl:allDifferent" are OWL classes for which there is

a mapping to a non-repeating list of their elements – though in fact, for a variety of reasons, the OWL semantics does not actually make this entirely explicit. The mapping itself is expressed by "owl:distinctMembers". To axiomatize these notions, we avail ourselves of the NOREPEATSLIST predicate introduced above.

```
(forall (L)
        (if (and (rdf:List L)
                 (NOREPEATSLIST L)
                 (forall (x)
                         (if (MEMBER x L)
                             (owl:Thing x))))
            (exists (C)
                    (and (owl:AllDifferent C)
                         (owl:distinctMembers C L)))))
```

Some discussion is in order here. The original intuition behind these two reserved predicates was to be able to indicate those classes whose members can be listed by a set of unique names. Classes for which there is such a listing are the members of owl:AllDifferent; and the lists to which a member of owl:AllDifferent are related by owl:distinctMembers are the lists that correspond to these listings, and hence are NOREPEATSLISTs. The semantics behind these intuitions would be fully captured by supplementing the above axiom with the following:[7]

```
(rdfs:subclassOf owl:AllDifferent owl:Class)

(forall (C L)
        (if (owl:distinctMembers C L)
            (and (owl:AllDifferent C)
                 (NOREPEATSLIST L)
                 (forall (x) (iff (C x)(MEMBER x L))))))
```

However, both axioms here imply that owl:Different is a subclass of owl:Class, i.e., a class of classes, and such classes are not permitted in OWL DL – the general semantics of OWL stipulates that all OWL classes be subclasses of owl:Thing, and a condition on OWL DL is that owl:Thing and owl:Class are disjoint – i.e., there are no classes of classes in OWL DL (see [OWL Semantics], Section 5.4). Hence, if the full semantics of these predicates as originally conceived were designed into OWL, they would be unavailable to OWL DL. The ability to specify non-repeating lists, however, was deemed too useful, so the fuller semantics expressed by the axioms above was simply omitted from the formal specification.

The upshot of this is that the semantic role of the members of owl:AllDifferent is reduced to that of simply "bare particulars" about which nothing can be said beyond that they have owl:distinctMember properties. Given the semantic vacuity of this role, it might seem to make the best sense simply to dispense with the class altogether and instead convert "owl:distinctMembers" to a class name denoting the class of nonrepeating lists.

---

[7] Thanks to Pat Hayes for this axiom and for filling out the historical and conceptual details surrounding "owl:AllDifferent" and "owl:distinctMembers".

OWL DL considerations once again come to the fore here, since classes of lists, like classes of classes, are also disallowed in OWL DL (Ibid.).

OWL Classification and Typing Axioms

Next we list a set of axioms that classify the central owl classes relative to themselves and those of RDFS. Note that, in virtue of the fact that everything is an rdfs:Resource, and that the domain of rdfs:subClassOf it follows that every OWL class is a subclass of rdfs:Resource.

```
(rdfs:subClassOf owl:Class rdfs:Class)

(rdfs:subClassOf rdfs:Datatype rdfs:Class)

(rdfs:subClassOf owl:ObjectProperty rdf:Property)

(rdfs:subClassOf owl:DatatypeProperty rdf:Property)

(rdfs:subClassOf owl:Restriction owl:Class)

(owl:disjointWith owl:ObjectProperty owl:DatatypeProperty x)
```

Domains and ranges

```
(rdfs:domain owl:distinctMembers owl:Thing)
(rdfs:range owl:distinctMembers owl:List)

(rdfs:domain owl:differentFrom owl:Thing)
(rdfs:range owl:differentFrom owl:Thing)

(rdfs:domain owl:disjointClass owl:Class)
(rdfs:range owl:disjointClass owl:Class)

(rdfs:domain owl:equivalentClass owl:Class)
(rdfs:range owl:equivalentClass owl:Class)

; A convenient definition

(forall (P)
        (iff (OWL_PROPERTY P)
             (or (owl:ObjectProperty P)
                 (owl:DatatypeProperty P))))

(rdfs:domain owl:equivalentProperty Owl_Property)
(rdfs:range owl:equivalentProperty Owl_Property)

(rdfs:domain owl:inverseOf Owl_Property)
(rdfs:range owl:inverseOf Owl_Property)

(rdfs:domain owl:sameAs owl:Thing)
(rdfs:range owl:sameAs owl:Thing)

(rdfs:domain owl:complementOf owl:Class)
```

```
(rdfs:range owl:complementOf owl:Class)

(rdfs:domain owl:intersectionOf owl:Class)
(rdfs:range owl:intersectionOf owl:Class)

(rdfs:domain owl:oneOf rdf:List)
(rdfs:range owl:complementOf owl:Class)

(rdfs:domain owl:unionOf owl:Class)
(rdfs:range owl:unionOf rdf:List)
```

Simple Arithmetic

OWL includes several primitives that involve natural numbers. Consequently, their intended semantics is best expressed axiomatically by including a bit of simple arithmetic, which we axiomatize here. Note that they are included for in order to provide a comprehensive account of OWL's intended semantics as a logical theory and not for automated reasoning. Most modern theorem provers include integers and arithmetical operations among their own primitives; hence, for purposes of automated reasoning, much better performance will be achieved by tuning the translation mechanisms to take sentences involving the numbers directly into sentences that call these arithmetical primitives directly, rather than by reasoning directly upon the axioms here.

OWL requires only a simple arithmetic with addition. We therefore add the constant '`0`', the function symbols '`+`' (addition) and '`+1`' (successor), and the predicate '`NatNum`' to our language, and add the following axioms:

```
; 0 is a natural number
(NatNum 0)

; The successor of a natural number is a natural number
(forall (n)
        (if (NatNum n)
            (NatNum (+1 n))))

; 0 plus any number n is n
(forall (n)
        (if (NatNum n)
            (= (+ n 0) n)))

; The sum of n and the successor of m is the successor of the sum of m
; and n
(forall (n m)
        (if (and (NatNum n) (NatNum m))
            (= (+ n (+1 m)) (+1 (+ n m)))))
```

It is useful to define the *less-than* relation on the natural numbers as well:

```
; m is less than n if m is the sum of n and some number other than 0
(forall (n m)
        (if (and (NatNum n) (NatNum m))
            (iff (< m n)
```

```
                (exists (r)
                        (and (NatNum r)
                             (not (= r 0))
                             (= n (+ m r)))))))
```

Basic Class and Property Theory for OWL

Logically speaking, OWL is first and foremost a theory of classes and binary relations, i.e., in the language of OWL, properties. This section provides the axioms that capture the basic features of these entities.

Intuitively, the class of owl:Things is the class of individuals. Exactly what individuals are taken to be is one of the things that distinguishes OWL Full ontologies from the other two types, as will be seen below.

```
; owl:Thing is an owl:Class  (This is derivable from the following
; axiom and the range axiom for owl:complementOf
(owl:Class owl:Thing)

; The class owl:Nothing is empty
(owl:complementOf owl:Thing owl:Nothing)

; owl:Classes only contain individuals
(forall (C)
        (if (owl:Class C)
            (subclass C owl:Thing))))

; owl:Classes are rdfs:Classes
(rdfs:subClassOf owl:Class rdfs:Class)

; Class comparison properties
(forall (C1 C2)
        (if (and (owl:Class C1) (owl:Class C2)
            (iff (owl:equivalentClasses C1 C2)
                 (forall (x) (iff (C1 x) (C2 x))))))

(forall (C1 C2)
        (if (and (owl:Class C1) (owl:Class C2)
            (iff (owl:disjointClasses C1 C2)
                 (forall (x) (not (and (C1 x) (C2 x)))))))
```

Property Axioms

```
; owl:FunctionalProperty
(forall (P)
        (iff (owl:FunctionalProperty P)
             (and (rdf:Property P)
                  (forall (x y z)
                          (if (and (P x y) (P x z))
                              (= y z))))))

; owl:InverseFunctionalProperty
(forall (P)
```

```
        (if (rdf:Property P)
            (iff (owl:InverseFunctionalProperty P)
                 (forall (x y z)
                         (if (and (P y x) (P z x))
                             (= y z))))))

; owl:equivlalentProperties
(forall (P1 P2)
        (if (and (rdf:Property P1) (rdf:Property P2))
            (iff (owl:equivalentProperties P1 P2)
                 (forall (x y)
                         (iff (P1 x y) (P2 x y))))))
```

Class Size

Given the simple arithmetic above, it is useful to introduce a notion of *size* for OWL classes. We do so recursively, making use in particular of the OWL primitive 'owl:unionOf':

```
(sizeOf owl:Nothing 0)

(forall (C1 C2 n x)
        (if (and (owl:Class C1)
                 (NatNum n)
                 (sizeOf C1 n)
                 (not (C1 x))
                 (owl:Class C2)
                 (forall (y) (iff (C2 y) (= y x))
             (sizeOf (owl:unionOf C1 C2) (+1 n))))
```

That is, if the size of class C1 is *n* and x is not an instance of C1, then the size of the union of C1 and {x} is *n*+1.

Class Operators

```
; complementOf
(forall (C1 C2)
        (if (and (owl:Class C1) (owl:Class C2))
            (iff (owl:complementOf C1 C2)
                 (forall (x)
                         (if (owl:Thing x)
                             (iff (C1 x) (not (C2 x))))))))

; unionOf
(forall (C L)
        (and (owl:Class C)
             (rdf:List L)
             (forall (Cls)
                     (iff (MEMBER Cls L) (owl:Class Cls))))
         (iff (owl:unionOf C L)
              (forall (x)
                      (iff (C x)
                           (exists (Cls)
                                   (and (MEMBER Cls L) (Cls x)))))))
```

```
; intersectionOf
(forall (C L)
        (and (owl:Class C)
             (rdf:List L)
             (forall (Cls)
                     (iff (MEMBER Cls L) (owl:Class Cls))))
         (iff (owl:intersectionOf C L)
              (forall (x)
                      (iff (C x)
                           (forall (Cls)
                                   (if (MEMBER Cls L) (Cls x)))))))

; oneOf
(forall (C L)
        (and (owl:Class C)
             (rdf:List L)
             (forall (x)
                     (or (iff (MEMBER x L) (owl:Thing x))
                         (iff (MEMBER x L) (rdfs:Literal x))))
             (iff (owl:one of C L)
                  (forall (x)
                          (iff (C x)
                               (exists (y)
                                       (and (MEMBER y L) (= x y))))))))
```

Comprehension Axioms for Class Operators

Since the class operations above are given simply as relations, there is no guarantee that unions, intersections, etc exist. Comprehension axioms are therefore required.

```
; Every class has a complement
(forall (C1)
        (if (owl:Class C1))
            (exists (C2)
                    (and (owl:Class C2)
                         (complementOf C1 C2)))))

; For every list of classes, there is a union of those classes
(forall (L)
        (if (and (rdf:List L)
                 (forall (C)
                         (if (MEMBER C L) (owl:Class C))))
            (exists (Cls)
                    (and (owl:Class Cls)
                         (owl:unionOf Cls L)))))

; For every list of classes, there is an intersection of those classes
(forall (L)
        (if (and (rdf:List L)
                 (forall (C)
                         (if (MEMBER C L) (owl:Class C))))
            (exists (C)
                    (and (owl:Class C)
                         (owl:intersectionOf C L)))))
```

```
; For every list of individuals, there is a class constaining exactly
; those individuals
(forall (L)
        (if (and (rdf:List L)
                 (forall (x)
                         (if (MEMBER x L) (owl:Thing x)))))
            (exists (C)
                    (and (owl:Class C)
                         (owl:oneOf C L)))))
```

OWL Restrictions

OWL restrictions are curious beasts. Ontologically speaking, restrictions are simply classes (as axiomatized already above). And in fact, every named class is also (coextensional with) a restriction.[8] And while the class of restrictions can be clearly and objectively defined, a class's being *considered* a restriction, in the context of an OWL ontology, is typically a highly contextualized affair. For the typical purpose of a restriction is in fact to place a condition upon membership in a class – in effect, to assert an axiom for it. The condition itself is given relative to a specified property. Thus, one always finds "owl:Restriction" paired with the "owl:onProperty" construct as well as a construct that specifies the precise nature of the restriction – typically, some cardinality or membership restriction. For example, the definition of the class Wine might look, in part, like this:

```
<owl:Class rdf:ID="Wine">
  <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasMaker" />
      <owl:someValuesFrom rdf:resource="#Winery" />
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
```
[9]

Thus, the construct introduces Wine as an owl:Class, and then declares this class to be a subclass of the restriction consisting of those things that bear the #hasMaker property to at least one winery. Thus, the effect of introducing the restriction in this context is to assert the axiom that every wine must have a maker.

---

[8] Specifically, any given class resource `Foo` is coextensional with the restriction

```
<owl:Restriction>
  <owl:onProperty rdf:resource="owl:sameAs">
  <owl:someValuesFrom rdf:resource="#Foo" />
</owl:Restriction>
```

[9] This is from [OWL Guide], section 3.4.1.

[POINT OUT that a restriction is a way of writing an axiom for a defined class that says, in effect: all members of the class being introduced have a certain characteristic. That is why it is typically used in conjunction with rdfs:subClassOf – to say that a class C is a subset of a restriction D is to say that every member of C has the characteristic represented by D.]

"owl:onProperty" and constraint conditions like "owl:someValuesFrom" can be given semantic values directly, but in fact they are artifacts stemming from the rather severe constraints of the underlying RDF syntax. Thus, despite the fact that they are separate elements, one can never find "owl:onProperty" occurring outside of the scope of a restriction; likewise every such occurrence is paired with either a *cardinality* constraint like "owl:someValuesFrom" or a *value* constraint. We can therefore translate the above restriction definition in terms of a new predicate 'Restriction_someVF', which is true of a class C1 (the "restriction" proper), a property P, and another class C2 just in case C1 consists of exactly those things that bear P to something in C2. To declare a given class C to be a subclass of C1 thus effectively "restricts" the way in which the members of C can bear P to other things. The semantics of this particular restriction is axiomatized thus:

```
(forall (C1 P C2)
   (iff (Restriction_someVF C1 P C2)
        (and (owl:Restriction C1)
             (rdf:Property P)
             (owl:Class C2)
             (forall (x)
                (iff (C1 x)
                     (exists (y)
                        (and (P x y)
                             (C2 y)))))))))
```

We introduce similar predicates – 'Restriction_allVF', 'Restriction_hasV' – for the other OWL restriction properties and axiomatize them accordingly.

```
(forall (C1 P C2)
   (iff (Restriction_allVF C1 P C2)
        (and (owl:Restriction C1)
             (rdf:Property P)
             (owl:Class C2)
             (forall (x)
                (iff (C1 x)
                     (forall(y)
                        (if (P x y)
                            (C2 y)))))))

(forall (C P v)
   (iff (Restriction_hasV C P v)
        (and (owl:Restriction C)
             (rdf:Property P)
             (owl:Thing v)
             (forall (x)
                (iff (C x)
                     (P x v))))))
```

The predicates ‘Restriction_minCard’ and ‘Restriction_maxCard’ for cardinality constrains are a bit more involved.

```
(forall (C P n)
   (iff (Restriction_minCard C P n)
        (and (owl:Restriction C)
             (rdf:Property P)
             (NatNum n)
             (forall (x)
                (iff (C x)
                     (exists (D)
                        (and (forall (y)
                                (if (D y) (P x y)))
                             (sizeOf D n)))))))))))
```

That is, C is a minimum cardinality restriction of *n* on property P iff C is an owl:Restriction, P an owl:Property, and *n* a natural number such that C is the class of all individuals x that bear P to at least *n* things (more exactly, the class of individuals x such that there is a class of things to which x bears P whose size is *n*). Similarly, when C is a maximum cardinality restriction of *n* on P, there can be no such class D of size *n*+1:

```
(forall (C P n)
   (iff (Restriction_maxCard C P n)
        (and (owl:Restriction C)
             (rdf:Property P)
             (NatNum n)
             (forall (x)
                (iff (C x)
                     (not (exists (D)
                          (and (forall (y)
                                  (if (D y) (P x y)))
                               (sizeOf D (+1 n)))))))))))
```

The ‘Restriction_Card’ predicate, which applies to cardinality restrictions requiring instances of a class C have exactly *n* distinct P-values, can now simply be defined in terms of the other cardinality restriction predicates:

```
(forall (C P n)
   (iff (Restriction_Card C P n)
        (and (Restriction_minCard C P n)
             (Restriction_maxCard C P n))))
```

A Comprehension Principle for Restrictions

It is now easy to state a relatively simple comprehension principle to the effect that there are all the various restrictions of the above types that there could be:

```
(forall (C2 P v n)
        (if (and (owl:Class C2)
                 (owl:Property P)
                 (owl:Thing v)
                 (NatNum n))
```

```
            (and (exists (C1) (Restriction_someVF C1 P C2))
                 (exists (C1) (Restriction_allVF C1 P C2))
                 (exists (C1) (Restriction_hasV C1 P C2))
                 (exists (C1) (Restriction_minCard C1 P n))
                 (exists (C1) (Restriction_maxard C1 P n))
                 (exists (C1) (Restriction_minCard C1 P n))
                 (exists (C1) (Restriction_Card C1 P n)))))
```

Note that with, with the axioms in the previous section, the odd property owl:onProperty falls out as a definable notion:

```
(forall (C1 P)
        (iff (owl:onProperty C1 P)
             (or (exists (C2) (Restriction_someVF C1 P C2))
                 (exists (C2) (Restriction_allVF C1 P C2))
                 (exists (v) (Restriction_hasV C1 P C2))
                 (exists (n) (Restriction_minCard C1 P n))
                 (exists (n) (Restriction_maxard C1 P n))
                 (exists (n) (Restriction_minCard C1 P n))
                 (exists (n) (Restriction_Card C1 P n)))))
```

OWL Ontologies

OWL is typically presented as coming in three flavors: OWL Lite, OWL DL, and OWL Full. The three flavors are often presented as three distinct *languages* with their own separate semantics that extend a common core. In fact, this is problematic, for reasons noted below. For purposes here, then, we will take OWL itself to be a single language with a single semantics and then characterize OWL *ontologies* as Lite, DL, or Full, as the case may be, depending on the constructs they require, allow, and rule out of court. It is easiest to start with OWL Full.

OWL Full Ontologies

The most distinctive semantic feature of OWL Full is that the primary semantic categories of OWL — things, properties, and classes — are assumed to be identical with their RDF(S) counterparts (see [OWL Semantics], 5.3). Thus, an OWL ontology is *Full* if only if it includes the following three axioms:

```
(= owl:Thing rdfs:Resource)

(= owl:ObjectProperty rdf:Property)

(= owl:Class rdfs:Class)
```

Note that from the fact that OWL datatype properties are RDF properties, i.e.,

```
; Theorem
(rdfs:subClassOf owl:DatatypeProperty rdf:Property)
```

it follows that OWL datatype properties are all OWL object properties in an OWL Full ontology:

```
(rdfs:subClassOf owl:DatatypeProperty owl:ObjectProperty) .
```

OWL DL Ontologies (THIS SECTION IS INCOMPLETE)

It is with OWL DL that one first sees why it is problematic to take OWL DL as a separate language. The problem is particularly vivid with regard to the notion of a *complex* property, as defined in [OWL Semantics]:

> To preserve decidability of reasoning in OWL Lite, not all properties can have cardinality restrictions placed on them or be specified as functional or inverse-functional. An individual-valued property is *complex* if 1/ it is specified as being functional or inverse-functional, 2/ there is some cardinality restriction that uses it, 3/ it has an inverse that is complex, or 4/ it has a super-property that is complex. Complex properties cannot be specified as being transitive.

Clearly, however, complexity is not intrinsic to a property; rather, it is an attribute that depends on context – notably, a given property might be referenced in a cardinality restriction in one ontology O1 but not in another O2. It would therefore be complex in the first but not in the second. A property is therefore complex or not *relative to* a given ontology. And an ontology is Full, DL or Lite depending on whether or not certain conditions hold. In particular, an ontology in which a property that is complex within that ontology is declared to be transitive cannot be DL; it can only be Full.

```
(forall (X Y)
   (if (and (or (= X owl:Thing) (= X owl:Datatype)
                (= X owl:ObjectProperty) (= X owl:DatatypeProperty)
                (= X rdf:List) (= X rdf:Literal)))
            (or (= Y owl:Thing) (= Y owl:Datatype)
                (= Y owl:ObjectProperty) (= Y owl:DatatypeProperty)
                (= Y rdf:List) (= Y rdf:Literal))
            (not (= X Y)))
       (owl:disjointWith X Y)))
```

An OWL ontology is *DL* if the following conditions hold.

- …

References

[RDF Semantics] http://www.w3.org/TR/2004/REC-rdf-mt-20040210

[Hayes-SW2CL] Hayes, P., Translating Semantic Web Languages into Common Logic, http://www.ihmc.us/users/phayes/CL/SW2SCL.html

[Hayes/Menzel]

[McG /Fikes] McGuinness, D., and R. Fikes, “An Axiomatic Semantics for RDF, RDF-S, and DAML+OIL”.