SWCLOS: A Semantic Web Processor on Common Lisp Object System

Seiji Koide
Galaxy Express Corporation
18-16, Hamamatsu-cho 1-chome, Minato-ku, Tokyo, Japan
koide@galaxy-express.co.jp

Masanori Kawamura
Galaxy Express Corporation
kawamura@galaxy-express.co.jp

Abstract

SWCLOS is a Semantic Web processor that is built on top of Common Lisp Object System (CLOS). Every resources in RDF and RDFS, *e.g.* rdfs:Class, rdfs:Resource, rdf:Property, and resource instances and properties are realized as CLOS objects with straightforward-mapping RDF/S classes to CLOS classes and RDF/S instances to CLOS instances. Axioms and entailment rules in RDF/S are embodied in the system so that a lisp programmer can codify ontology in RDF/S and use the ontology within the semantics specified by RDF/S documents. SWCLOS can read and write RDF/XML and N-triples format files as well as S-expression files. Thus, lisp programmers with SWCLOS can enjoy RDF/S programming in S-expression from the beginning to the end in their work without touching XML in communication with other people. In this paper, some examples same as on the Jena tutorial are demonstrated for the introduction of SWCLOS programming in comparison with Java, and a demonstration with the wine ontology explains the domain and range constraint functionality in SWCLOS. SWCLOS is opened to the public in the BSD-like Open Source principle. Contact the web page or the above email address.

Keywords

Markup and Data Formats, Semantic Webs, RDF, RDFS, OWL, Axioms, Entailments, CLOS, OOP

1. Introduction

Object-Oriented Programming is an appropriate approach for Semantic Web processing as well as ordinary data processing. Although many Semantic Web toolkits are programmed with object-oriented languages such as Python, Java, and C#, they cannot let classes and objects in program code directly coincide with RDFS and OWL classes and instances. The reason is that those programming languages do not provide full-bodied *reflective* programming functionality. Here the term "*reflective*" indicates not only the capability to dynamically identify a procedural method in context and invoke the method in run-time, but also the capability to modify the language's implementation without leaving the realm of the language [1].

RDFS and OWL are reflective languages. Namely, Classes in RDFS are themselves resources [2]. The members of a class are known as instances of the class. It is noted that rdfs:Class is a universal class [3], that is a class of any class. An instance of rdfs:Class is itself. In OWL Full, a class can be treated simultaneously as a collection of individuals and as an individual in its own right [4]. OWL Full requires classes to be treated as individuals. Such an idea is called *reflection*.

Reflective Knowledge Representation (KR) and reflective programming has been explored in two decades [5, 6, and 7]. From the viewpoint of KR, self-reference is a key issue required for conducting meta-theory and coping with cognitive overflow [6]. From the viewpoint of programming, meta-circularity is a key technology to enable meta-modeling. 3-Lisp [5] was the first reflective programming language, using `eval` and `apply` in lisp, but it was not object-oriented.

According to the idea of reflection, Common Lisp Object System (CLOS) was designed to specify a model for the language implementation and to standardize it. Thus, a programmer can manipulate the internal working machinery in language processing using CLOS Meta-Object Protocol (MOP). Therefore, it is conceivable to implement RDFS and OWL reflective functionality on top of CLOS.

We have developed an Object-Oriented Semantic Web processor on top of CLOS with straightforward mapping RDFS instances onto CLOS instances, RDFS classes onto CLOS classes, RDFS meta-classes or rdfs:Class and rdfs:Datattype, onto CLOS meta-classes. In this paper, firstly we discuss RDF syntax in S-expression; secondly we demonstrate the semantics and the availability of SWCLOS as RDFS processor. Furthermore we discuss the extension to OWL, and finally exploit a simple example with reflective programming. Eventually CLOS reflective programming functionality for Semantic Webs will be fully available in OWL Full realization.

2. RDF Syntax in SWCLOS

RDF/XML syntax is firmly specified in the Syntax Specification [8]. SWCLOS can read RDF/XML format files and translates contents in XML into S-expression that is similar to RDF/XML syntax. In this section, we state the equality between both representations

including blank nodes.

2.1 Entities in SWCLOS

In SWCLOS, a resource in RDF graph is a named CLOS object whose name is the same as QName or nodeID. A QName and a nodeID in SWCLOS is a lisp symbol and a resource object is bound to the QName or nodeID symbol. A resource object may have slots or pairs of property name (slot role) and property value (slot filler), that are pairs of arc label and pointed entity in RDF graphs. A simple literal in SWCLOS is interpreted as either string or number in lisp, and a typed literal is also interpreted as appropriate instance of corresponding type, *e.g.*, xsd:nonNegativeInteger or xsd:float, *etc*., that are lisp types defined in SWCLOS. All of entities in the RDFS specification are intrinsically embodied in SWCLOS. The following demonstrates an example of embodied entity.

```
gx-user(7): rdfs:seeAlso
#<rdf:Property rdfs:seeAlso>
gx-user(8): (describe rdfs:seeAlso)
#<rdf:Property rdfs:seeAlso> is an instance of #<rdfs:Class rdf:Property>:
 The following slots have :instance allocation:
  name              rdfs:seeAlso
  about             #<uri http://www.w3.org/2000/01/rdf-schema#seeAlso>
  label             "seeAlso"
  comment           "Further information about the subject resource."
  isDefinedBy       #<uri http://www.w3.org/2000/01/rdf-schema>
  mclasses          (#<rdfs:Class rdf:Property>)
  domain            #<rdfs:Class rdfs:Resource>
  range             #<rdfs:Class rdfs:Resource>
  subPropertyOf     <unbound>
  superPropertyOf   (#<rdf:Property rdfs:isDefinedBy>)
```

2.2 S-expression for RDF

An example of RDF graph in the Jena tutorial [9] is represented in S-expression as follows. The macro `defIndividual` allows users to define a resource instance.

```
gx-user(9): (defpackage vCard
              (:documentation "http://www.w3.org/2001/vcard-rdf/3.0#"))
#<The vCard package>
gx-user(10): (defpackage somewhere (:documentation "http://somewhere/"))
#<The somewhere package>
gx-user(11): (defIndividual somewhere::JohnSmith
               (rdf:about "http://somewhere/JohnSmith")
               (vCard::FN "John Smith")
               (vCard::N (rdfs:Resource (vCard::Given "John")
                                        (vCard::Family "Smith"))))
Warning: Entail: vCard::FN rdf:type rdf:Property
Warning: Entail: vCard::N rdf:type rdf:Property
Warning: Entail: vCard::Given rdf:type rdf:Property
Warning: Entail: vCard::Family rdf:type rdf:Property
#<rdfs:Resource somewhere:JohnSmith>
gx-user(12): somewhere:JohnSmith
#<rdfs:Resource somewhere:JohnSmith>
gx-user(13): (get-form somewhere:JohnSmith)
(rdfs:Resource somewhere:JohnSmith
  (rdf:about #<uri http://somewhere/JohnSmith>)
  (vCard:FN "John Smith")
  (vCard:N (rdfs:Resource (vCard:Family "Smith") (vCard:Given "John"))))
gx-user(14): (write-xml somewhere:JohnSmith *standard-output*)
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#"
         xmlns:somewhere="http://somewhere/" >
  <rdf:Description rdf:about="http://somewhere/JohnSmith" >
    <vCard:FN>John Smith</vCard:FN>
    <vCard:N>
      <rdf:Description>
        <vCard:Family>Smith</vCard:Family>
        <vCard:Given>John</vCard:Given>
      </rdf:Description>
```

```
    </vCard:N>
  </rdf:Description>
</rdf:RDF>
```

Here your eyes may catch several warning for entailments at first. The entailment functionality in SWCLOS is explained later. In this example, "vCard" and "somewhere" namespaces are created by lisp package definitions with an optional document parameter for the URIs. Then, the graph is inputted in S-expression that constructs similar nested syntax structure to the structure in RDF/XML. A lisp symbol `somewhere::JohnSmith` is designated as this resource QName or CLOS object name for the URI "http://somewhere/JohnSmith". Note that you must attach double colons instead of single colon of QName, when you input QNames into SWCLOS at first. Once the system knows a QName, you may refer it with single colon just like QName appearance. The created resource object is bound to the QName, here `somewhere:JohnSmith`. So, typing the QName symbol in top level window returns the bound resource object. Function `get-form` returns the S-expression form for the object in the sense of RDF. At last, function `write-xml` prints the information of the resource in RDF/XML.

2.3 Blank nodes in RDF

The above example also illustrates blank node expression in SWCLOS. A blank node that is pointed via an arc by another node in RDF graphs is represented as a property value that is expressed by a list composed of a type designator, no name, and slots, as demonstrated at the value of property "`vCard::N`". There is no way in SWCLOS to represent a blank node for a root of RDF graph trees. A node that has a URI is identified globally; therefore the descriptions of nodes that have same URI are unified from different information sources. However, blank nodes cannot be unified even though they are composed of same slot structures with same information. Therefore, the following example demonstrates basic principles of information adding in RDF. Suppose inputting the same thing as before again, see the result.

```
gx-user(14): (defIndividual somewhere:JohnSmith
               (rdf:about "http://somewhere/JohnSmith")
               (vCard:FN "John Smith")
               (vCard:N (rdfs:Resource (vCard:Given "John")
                                       (vCard:Family "Smith"))))
#<rdfs:Resource somewhere:JohnSmith>
gx-user(15): (get-form somewhere:JohnSmith)
(rdfs:Resource somewhere:JohnSmith
  (rdf:about #<uri http://somewhere/JohnSmith>)
  (vCard:FN "John Smith")
  (vCard:N (rdfs:Resource (vCard:Family "Smith") (vCard:Given "John"))
           (rdfs:Resource (vCard:Family "Smith") (vCard:Given "John"))))
```

The triple of `somewhere:JohnSmith/vCard:FN/"John Smith"` is not added, because same QNames and same literals are unified and the triple is captured as same in set. However, the triple of `somewhere:JohnSmith/vCard:N/<the second blank node>` cannot be deemed to be same as set, because both blank nodes cannot be unified.

3. RDFS Semantics in SWCLOS

3.1 Instances, Classes, Meta-classes, and Axioms

Figure 1 summarizes the relationship of rdf:type, rdfs:subClassOf, and rdfs:subPropertyOf among RDF Schema entities. We have mapped rdfs:type to CLOS class-instance relationship and mapped rdfs:subClassOf to CLOS super-subclass relationship. Such a straightforward mapping yields various axioms on type and subsumption, which are described in RDF Semantics [3], among CLOS objects without any dedicated programming, because CLOS objects have same class-instance semantics on type and subsumption. For instance, the following shows RDF axioms using lisp native type predicate `cl:typep`.

```
gx-user(16): (cl:typep rdf:type rdf:Property)          → t
gx-user(17): (cl:typep rdf:subject rdf:Property)       → t
gx-user(18): (cl:typep rdf:predicate rdf:Property)     → t
gx-user(19): (cl:typep rdf:object rdf:Property)        → t
gx-user(20): (cl:typep rdf:first rdf:Property)         → t
gx-user(21): (cl:typep rdf:rest rdf:Property)          → t
gx-user(22): (cl:typep rdf:value rdf:Property)         → t
gx-user(23): (cl:typep rdf:_1 rdf:Property)            → t
gx-user(24): (cl:typep rdf:_2 rdf:Property)            → t
gx-user(25): (cl:typep rdf:nil rdf:List)               → t
```

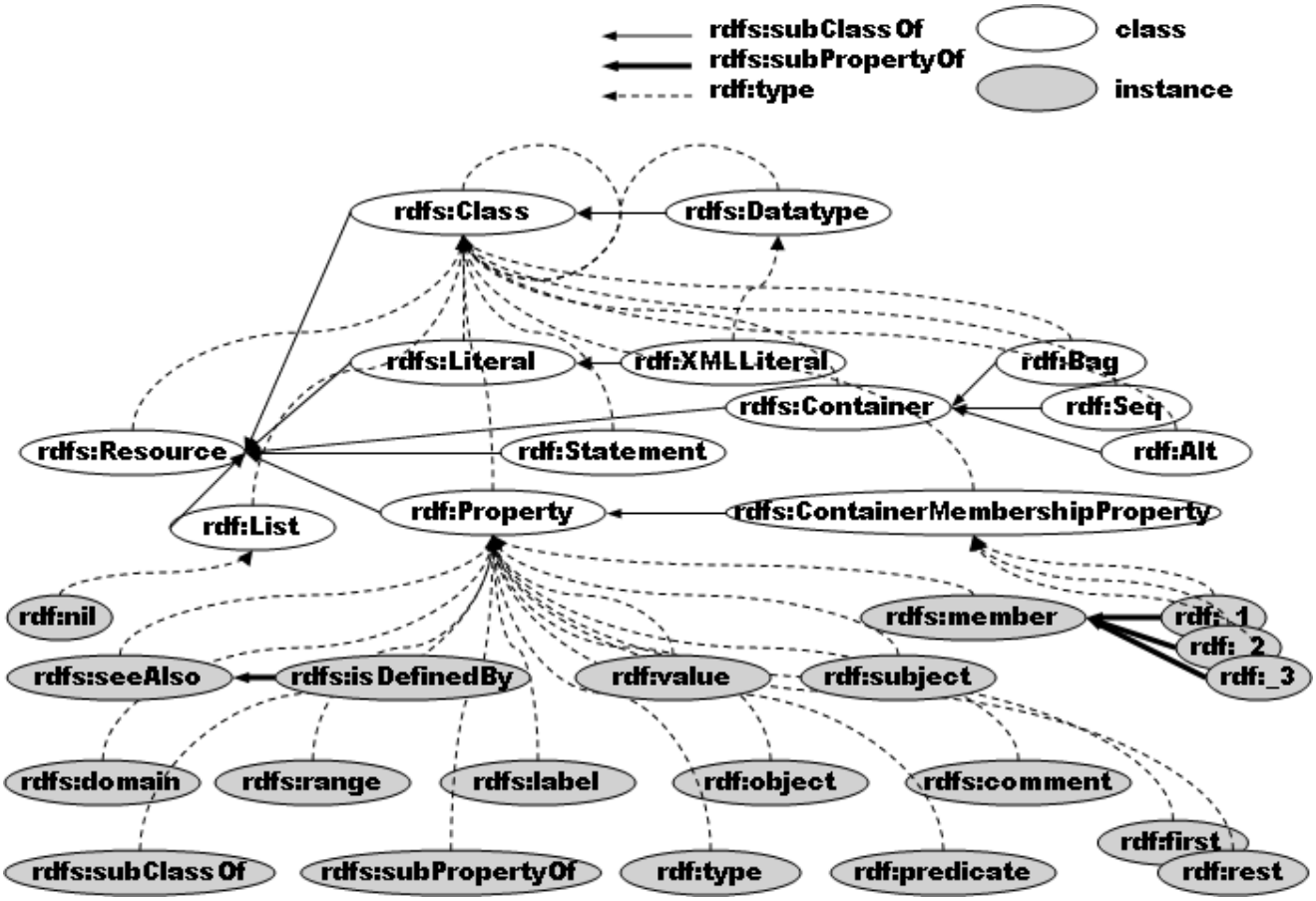

Figure 1. Hierarchical Structure in RDFS

RDFS axioms described in RDF Semantics [3] except the domain and range axioms are also automatically realized by CLOS as follows using lisp native `cl:typep` and `cl:subtypep`.

```
gx-user(27): (cl:subtypep rdf:Alt rdfs:Container)                          → t t
gx-user(28): (cl:subtypep rdf:Bag rdfs:Container)                          → t t
gx-user(29): (cl:subtypep rdf:Seq rdfs:Container)                          → t t
gx-user(30): (cl:subtypep rdfs:ContainerMembershipProperty rdf:Property)→ t t
gx-user(31): (cl:typep rdf:XMLLiteral rdfs:Datatype)                       → t
gx-user(32): (cl:subtypep rdf:XMLLiteral rdfs:Literal)                     → t t
gx-user(33): (cl:subtypep rdfs:Datatype rdfs:Class)                        → t t
gx-user(34): (cl:typep rdf:_1 rdfs:ContainerMembershipProperty)            → t
gx-user(35): (cl:typep rdf:_2 rdfs:ContainerMembershipProperty)            → t
```

The other axioms that are derived the above axioms and the domain and range axioms, which are described in the document [3], are also realized as well, except the relation between rdfs:Class as the universal class and rdfs:Class as its extension.

Figure 1 shows that every class except rdfs:Class and rdfs:Datatype is an instance of either rdfs:Class or rdfs:Datatype, and oppositely the two are classes of the other classes. Such a class, a class of classes, is called a *metaclass* in CLOS [10]. In Figure 1, the rdf:type property of rdfs:Class points itself. This circularity in type is called *meta-circularity* [1]. The metaclassing is crucial enabling technology in reflective programming. Instances are manipulated with the class methods. The behaviors of classes, *e.g.* instantiate method etc., are controlled by the metaclass methods. The CLOS Meta-Object Protocol (MOP) allows us to change the behaviors of classes through the methods of metaclasses. Although ideally multiple layering could be infinite like the base (instance) layer, the class layer, the metaclass layer, the meta-metaclass layer and so on, actually the infinity is terminated by the meta-circularity or self-reference as shown at rdfs:Class.

Mapping rdfs:Class and rdfs:Datatype onto CLOS metaclasses resulted from mapping rdf:type to CLOS class-instance relationship. This causes a difficult problem. An instance is created by its class, however when rdfs:Class object is about to be instantiated, there is no class as mother. This problem is same as in CLOS implement on `standard-class`, and the bootstrapping machinery in CLOS was developed [11]. The lessons learned from the implementation on reflective programming languages should be helpful to implement RDFS and OWL reflection in the case that the processor is developed from scratch. However, in this case SWCLOS has been developed on top of CLOS, a reflective system, thus the question was how to implement a reflective system on top of another reflective system. The detail of the implementation will be described in another document. Anyway the relation between rdfs:Resource and rdfs:Class and the relation between rdfs:Class and rdfs:Datatype were somehow carried out. However it was impossible to avoid causing an error in specifying directly meta-circularity on rdfs:Class by the implementation language, *i.e.* Allegro Common Lisp. So, the meta-circularity on

rdfs:Class is pretended by introducing a customized type predicate `gx:typep`.

```
gx-user(36): (cl:typep rdfs:Resource rdfs:Class)                    → t
gx-user(37): (cl:typep rdfs:Datatype rdfs:Class)                    → t
gx-user(38): (cl:subtypep rdfs:Class rdfs:Resource)                 → t t
gx-user(39): (cl:subtypep rdfs:Resource rdfs:Resource)              → t t
gx-user(40): (gx:typep rdfs:Class rdfs:Class)                       → t
```

3.2 Subsumption Entailment

The rdf:type and the subsumption entailment also resulted from the straightforward mapping RDFS schema to CLOS objects. **Figure 2** illustrates a part of Wine Ontology in RDFS that is modified from OWL [12]. The lisp native type predicate carries out the subsumption entailment (RDFS entailment rule **rdfs9** in RDFS model theory [3]) as demonstrated for the Wine Ontology as follows.

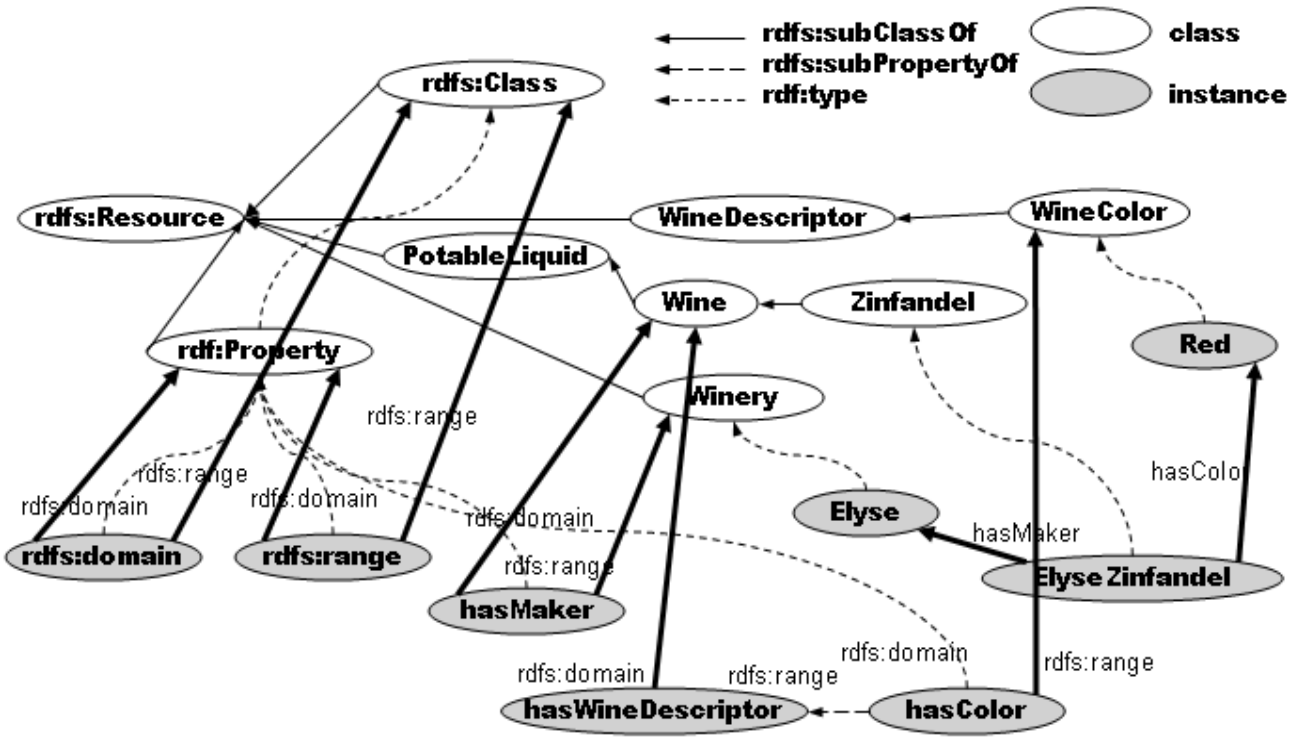

Figure 2. Wine Ontology in RDFS

```
gx-user(5): (defpackage vin (:documentation "http://somewhere/wine.rdf"))
#<The vin package>
gx-user(6): (defpackage food (:documentation "http://somewhere/food.rdf"))
#<The food package>
gx-user(7): (/. vin::Wine rdfs:subClassOf food::PotableLiquid)
Warning: Entail in vin::Wine rdfs:subClassOf food::PotableLiquid:
..... vin::Wine rdf:type rdfs:Class.
Warning: Entail in #<rdfs:Class vin:Wine> rdfs:subClassOf food::PotableLiquid:
..... food::PotableLiquid rdf:type rdfs:Class.
#<rdfs:Class vin:Wine>
gx-user(8): (/. vin::Zinfandel rdfs:subClassOf vin:Wine)
Warning: Entail in vin::Zinfandel rdfs:subClassOf #<rdfs:Class vin:Wine>:
..... vin::Zinfandel rdf:type rdfs:Class.
#<rdfs:Class vin:Zinfandel>
gx-user(9): (/. vin::ElyseZinfandel rdf:type vin:Zinfandel)
#<vin:Zinfandel vin:ElyseZinfandel>
```

Then, the predicate `cl:typep` carries out the subsumption entailment rule **rdfs9** as follows.

```
gx-user(12): (cl:typep vin:ElyseZinfandel vin:Wine)          → t
```

```
gx-user(13): (cl:typep vin:ElyseZinfandel food:PotableLiquid) → t
```

The transitivity entailment (RDFS entailment rule **rdfs11**) is also demonstrated by predicate `cl:subtype` as follows.

```
gx-user(14): (cl:subtypep vin:Zinfandel food:PotableLiquid) → t
```

3.3 Domain and Range of Property and the Entailment

RDFS is a property centric language. Namely, properties are firstly defined with the domain and range constraint definitions of the property. SWCLOS attempts to satisfy domain and range constraints when a triple inputted. In the case that a subject and/or an object in a triple are not defined, SWCLOS defines them as instances of the constraining classes and data types (RDFS entailment rule **rdfs2** and **rdfs3**).

For instance, in the previous section, undefined food:PotableLiquid, vin:Wine, and vin:Zinfandel were defined as instances of rdfs:Class (it implies they are classes), since the domain and range of rdfs:subClassOf is rdfs:Class. In the following example, the first input causes the entailment reasoning of the domain constraint for rdfs:domain, the second input carries out the range constraint entailment for rdfs:range, and the last input entailes vin:Red is a vin:WineColor because of the range constraint of vin:hasColor.

```
gx-user(19): (/. vin::hasColor rdfs:domain vin:Wine)
Warning: Entail in vin::hasColor rdfs:domain #<rdfs:Class vin:Wine>:
..... vin::hasColor rdf:type rdf:Property.
#<rdf:Property vin:hasColor>
gx-user(20): (/. vin:hasColor rdfs:range vin::WineColor)
Warning: Entail in #<rdf:Property vin:hasColor> rdfs:range vin::WineColor:
..... vin::WineColor rdf:type rdfs:Class.
#<rdf:Property vin:hasColor>
gx-user(21): (/. vin:ElyseZinfandel vin:hasColor vin::Red)
Warning: Entail in #<vin:Zinfandel vin:ElyseZinfandel> vin:hasColor vin::Red:
..... vin::Red rdf:type vin:WineColor.
#<vin:Zinfandel vin:ElyseZinfandel>
```

This domain and range entailment couples with the subsumption entailment. The extensional entailments (**ext1** and **ext2**) are carried out in predicates `domainp` and `rangep` in SWCLOS. For instance, see the followings.

```
gx-user(30): (domainp vin:hasColor vin:Wine)          → t
gx-user(31): (rangep vin:hasColor vin:WineColor)      → t
gx-user(32): (subtypep vin:Wine food:PotableLiquid)   → t
gx-user(33): (/. vin:WineColor rdfs:subClassOf vin::WineDescriptor)
Warning: Entail in #<rdfs:Class vin:WineColor> rdfs:subClassOf vin::WineDescriptor:
..... vin::WineDescriptor rdf:type rdfs:Class.
#<rdfs:Class vin:WineColor>
gx-user(34): (domainp vin:hasColor food:PotableLiquid) → t
gx-user(35): (rangep vin:hasColor vin:WineDescriptor)  → t
```

The domain and range of vin:hasColor is subsumed by food:PotableLiquid and vin:WineDescriptor, because vin:Wine is a subclass of food:PotableLiquid and vin:WineColor is a subclass of vin:WineDescriptor.

3.4 rdfs:subPropertyOf

In ordinary object systems, classes have super-subclass relationship, but there is no super-sub relationship among instances. However, there is rdfs:subPropertyOf on property in RDFS, while a property is an instance of rdf:Property class. The rdfs:subPropertyOf functionality conveys the subsumption on property (RDFS extensional entailment rule **ext3**, **ext4**). SWCLOS provides retrieval functions `get-domain` and `get-range` that inherit domain or range information defined at the super-properties.

For instance, see the following example. Note that this is done after rebooting SWCLOS, so that the ontology illustrated in Figure 2 is realized.

```
gx-user(5): (defpackage vin (:documentation "http://somewhere/wine.rdf"))
..... ditto .....
gx-user(6): (defpackage food (:documentation "http://somewhere/food.rdf"))
..... ditto .....
gx-user(7): (/. vin::Wine rdfs:subClassOf food::PotableLiquid)
..... ditto .....
gx-user(8): (/. vin::Zinfandel rdfs:subClassOf vin:Wine)
..... ditto .....
gx-user(9): (/. vin::ElyseZinfandel rdf:type vin:Zinfandel)
```

```
..... ditto .....
gx-user(10): (/. vin::hasWineDescriptor rdfs:domain vin:Wine)
Warning: Entail in vin::hasWineDescriptor rdfs:domain #<rdfs:Class vin:Wine>:
..... vin::hasWineDescriptor rdf:type rdf:Property.
#<rdf:Property vin:hasWineDescriptor>
gx-user(11): (/. vin::hasColor rdfs:subPropertyOf vin:hasWineDescriptor)
Warning: Entail in vin::hasColor rdfs:subPropertyOf #<rdf:Property vin:hasWineDescriptor>:
..... vin::hasColor rdf:type rdf:Property.
#<rdf:Property vin:hasColor>
gx-user(12): (/. vin::WineColor rdfs:subClassOf vin::WineDescriptor)
Warning: Entail in vin::WineColor rdfs:subClassOf vin::WineDescriptor:
..... vin::WineColor rdf:type rdfs:Class.
Warning: Entail in #<rdfs:Class vin:WineColor> rdfs:subClassOf vin::WineDescriptor:
..... vin::WineDescriptor rdf:type rdfs:Class.
#<rdfs:Class vin:WineColor>
gx-user(13): (/. vin:hasColor rdfs:range vin::WineColor)
#<rdf:Property vin:hasColor>
gx-user(14): (get-domain vin:hasColor)            → #<rdfs:Class vin:Wine>
gx-user(15): (domainp vin:hasColor vin:Wine)      → t
gx-user(16): (get-range vin:hasColor)             → <rdfs:Class vin:WineColor>
gx-user(17): (rangep vin:hasColor vin:WineDescriptor)  → t
```

Here `get-domain` returns vin:Wine that is inherited from the super-property vin:hasWineDescriptor. The predicate `domainp` also uses the inherited value. While `get-range` retrieved vin:WineColor that is directory taken from the range property of vin:hasColor, predicate `rangep` can reach vin:WineDescriptor from vin:hasColor.

3.5 Monotonicity and Forward Reference

Forward reference in input must be allowable. Fortunately the monotonicity of knowledge is required by RDFS theory. In other words, knowledge must be added and refined more and more monotonically. The pieces of knowledge must not be deleted, and the ontology must not be reduced back to obscure abstract states. Therefore, we can generate forward-referenced objects in the sense at that time, when we encounter undefined objects. The various axioms and entailments help us to decide which class should be applied to the object creation. For example, in the section 2.2, undefined vCard properties are defined as an instance of rdf:Property by the entailment rule **rdfs1** (a predicate in triple entails it is a property). When more exact class definitions are stated lately, re-definitions for classes may be performed. If a relevant but broader sense is stated after precise meanings, it has no effect. If a irrelevant notion is stated, it should be added.

The **rdfs4a** and **rdfs4b** (entailment that any referent as subject or object but not property is an instance of rdfs:Resource) accepts a reference that is not qualified with rdf:type, rdfs:subClassOf, and any constraint of domain or range. However it embraces the ambiguity that the referenced object may be an instance (as instance of rdfs:Resource), or a class (as instance of rdfs:Class, remind that rdfs:Class is a subclass of rdfs:Resource, in RDFS everything is a resource including classes and metalcasses). For the case, the referent is tentatively defined as an instance of rdfs:Resource (namely instance), and it is to be redefined when it is stated to be a class.

4. Extension to OWL

4.1 Local Property Range Constraint

We are now extending SWCLOS to OWL. OWL is more likely object-oriented than RDFS. SWCLOS based on object-oriented system basically fits OWL much more than RDFS. In OWL each class may have various local constraints for the property value, *i.e.*, cardinality, value type, and value. In CLOS, the slot-definitions for slots in each instance object is stored in the class of the instance for housekeeping. The slot-definition object has a type of instance slot value, which may be a localized property value constraint. In practice, SWCLOS instantiates slot-definition objects whose slot-value types origin from the property ranges in RDFS, when the object is instantiated. The range checking and entailing is performed consulting with the slot-value type in the slot-definition object, when the instance is created. In OWL, the local property range constraint that is expressed by owl:onProperty and owl:allValuesFrom, or owl:hasValue is stored in the slot-definitions as well as the range constraint in RDFS. Thus, SWCLOS can carry out the entailment of property range locally in OWL as well as globally in RDFS.

The following example demonstrates the entailment from owl:allValuesFrom constraint. Function `read-rdf-file` parses RDF/XML file, and function `addRdfXml` interprets the parsed result.

```
gx-user(2): (read-rdf-file #'addRdfXml "food.rdf")
      ; many warning for entailment.
gx::done
gx-user(3): (read-rdf-file #'addRdfXml "wine.rdf")
      ; many warning for entailment.
gx::done
```

```
gx-user(4): (get-form vin:Wine)
(owl:Class vin:Wine (rdfs:label "wine" "vin")
 (rdfs:subClassOf food:PotableLiquid
                  (owl:Restriction
                   (owl:cardinality 1)
                   (owl:onProperty #<owl:FunctionalProperty vin:hasMaker>))
                  (owl:Restriction
                   (owl:allValuesFrom #<owl:Class vin:Winery>)
                   (owl:onProperty #<owl:FunctionalProperty vin:hasMaker>))
                  (owl:Restriction
                   (owl:minCardinality 1)
                   (owl:onProperty #<owl:ObjectProperty vin:madeFromGrape>))
                  (owl:Restriction
                   (owl:cardinality 1)
                   (owl:onProperty #<owl:FunctionalProperty vin:hasSugar>))
                  (owl:Restriction
                   (owl:cardinality 1)
                   (owl:onProperty #<owl:FunctionalProperty vin:hasFlavor>))
                  (owl:Restriction
                   (owl:cardinality 1)
                   (owl:onProperty #<owl:FunctionalProperty vin:hasBody>))
                  (owl:Restriction
                   (owl:cardinality 1)
                   (owl:onProperty #<owl:FunctionalProperty vin:hasColor>))
                  (owl:Restriction
                   (owl:onProperty #<owl:TransitiveProperty vin:locatedIn>)
                   (owl:someValuesFrom #<owl:Class vin:Region>))))
gx-user(5): (defIndividual MyHomeMadeWine (rdf:type vin:Wine) (vin:hasMaker MyHome))
Warning: Entail by rdfs4b: MyHome rdf:type rdfs:Resource.
Warning: Entail:change class of #<rdfs:Resource MyHome> to #<owl:Class vin:Winery>.
#<vin:Wine MyHomeMadeWine>
```

At first, the food.rdf [13] and the wine.rdf file [12] in OWL were read into the system. Then the inputted vin:Wine form was demonstrated in S-expression as shown in the example. At last, my homemade wine was defined as individual with the slot (vin:hasMaker MyHome). SWCLOS firstly entailed that MyHome is a resource, then entailed it should be a vin:Winery, because vin:hasMaker in vin:Wine must have an instance of vin:Winery from the owl:allValuesFrom constraint.

4.2 Cardinality Constraint

Cardinality constraint is also implemented by introducing new facets of property, gx::maxcardinality and gx::mincardinalty, into the slot-definition. SWCLOS just checks the number of values in a slot, whenever a slot value is set or added into a slot.

5. Exploring Reflective Programming

Reflective programming in RDFS and OWL is an unexplored and challenging theme in Semantic Web programming. We demonstrate a trivial exmaple for handling classes as they are instances using SWCLOS. Suppose that we have OWL-S simple processes and atomic processes [14] that we want to deal with them as if they were simple process classes and atomic process classes. Note that a simple process is an instance of process:SimpleProcess and an atomic process is an instance of process:AtomicProcess. We can make this kind of special objects by introducing metaclasses for SimpleProcess and AtomicProcess and setting up them as shown in **Figure 3**.

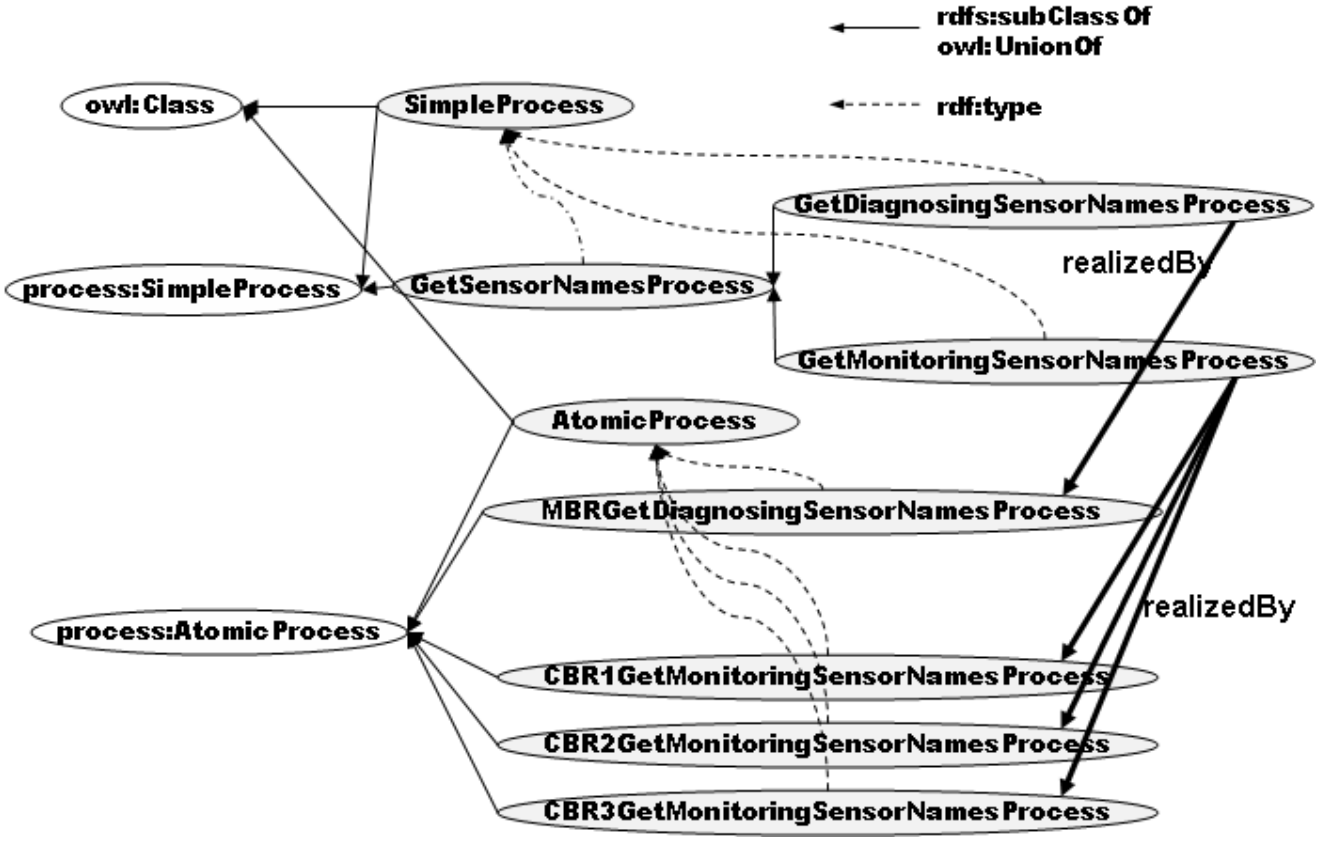

Figure 3. An Example of Reflective Programming

Such setting allows those simple processes to have both rdfs:subClassOf property and owl:realizedBy property, and allows those atomic processes to have both property range constraint for instance objects and owl:describe property.

References

[1] Paepcke, A.: *User-Level Language Crafting Introducing the CLOS Metaobject Protocol*, Paepcke, A. (*ed.*), Object-Oriented Programming - The CLOS Perspective, MIT Press, (1993) 65-99.

[2] Brickley D., R.V. Guha (*ed.*): *RDF Vocabulary Description Language 1.0: RDF Schema*, http://www.w3.org/TR/rdf-schema/

[3] Hayes, P., B. McBride, (*ed.*): *RDF Semantics*, http://www.w3.org/TR/rdf-mt/

[4] Bechhofer, S., *et al.* (*ed.*): *OWL Web Ontology Language Reference*, http://www.w3.org/TR/owl-ref/#OWLFull

[5] Smith, B.: *Reflection and Semantics in Lisp*, Proc. 1984 ACM Principles of Programming Language Conference, ACM, (1984) 23-35

[6] Weyhrauch, R.W.: *Prolegomena to a Theory of Mechanized Formal Reasoning*, Artificial Intelligence, Vol.13. (1980) 133-170

[7] Bowen, K.: *Meta-level Techniques in Logic Programming*, Proc. Int. Conf. AI and its Applications, (1986)

[8] Beckett, D.: *RDF/XML Syntax Specification (Revised)*, http://www.w3.org/TR/rdf-syntax-grammar/

[9] http://jena.sourceforge.net/tutorial/RDF_API/index.html

[10] Steele Jr., G.L.: *Common Lisp The Language second edition*, Digital Press (1990)

[11] Kiczales, G., des Rivières, J., Bobrow, D.G.: *The Art of the Metaobject Protocol*, MIT Press, (1992)

[12] http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf

[13] http://www.w3.org/TR/2004/REC-owl-guide-20040210/food.rdf

[14] http://www.daml.org/services/owl-s/1.1B/