Meta-Object Protocol in SWCLOS

Nov. 23, 07

Seiji Koide

SWCLOS, a Semantic Web processor on top of CLOS, is developed by leveraging the original Meta-Object Protocol of Common Lisp Object System (CLOS), which allows us reflective programming. RDF Schema (RDFS) is a semantic extension of RDF, and OWL is also an extension of RDF. The fundamental model of RDF, eventually of RDFS and OWL, is graphs. Therefore, basic semantics of RDF, RDFS, and OWL differs from CLOS, and the functionalities of their processors should be different from original CLOS. Specifically, nodes in graphs are freely added into a set of graphs without the definition of their classes (every node of graphs in the RDF universe is an instance of rdfs:Resource) and new edges in directed graphs can connect any two nodes (every edge of graphs in the RDF universe constructs an extension of the property, namely a tuple of the relation or the property), while a class of instances in CLOS must be established before instance creation and the slot definitions must be established before slot value bindings. In SWCLOS, such semantics in RDF(S) and OWL is established in the realms of CLOS by virtue of the Meta-Object Protocol (MOP) of CLOS. In this document, I summarize the Meta-Object Protocol (MOP) in SWCLOS, or how the original MOP is extended for Semantic Webs, RDF(S), and OWL.

Entailment Rules and Principle of Monotonicity

Fortunately, we have the principle of monotonicity and several rules that are available to specify the class of instances of which class is not specified.

Default Metaclass in RDF

In the semantics of Object-Oriented Programming, a class is a prototype of its instances and we need the definition of class before making an object. In case that any instance is defined as entity in the RDF universe, earlier than its class definition, we can suppose that the class of instances is an instance of metaclass rdfs:Class, because every class in the RDF universe is an instance of rdfs:Class. Therefore, we make the following method to do so.

```
(defmethod make-instance :before ((class-name symbol) &rest initargs)
  (declare (dynamic-extent initargs))
  (cond ((find-class class-name cl:nil)) ; nothing done
        (t (mop:ensure-class class-name :metaclass rdfs:Class))))
```

On the other hand, you must designate a metaclass in class definition, if you direct any metaclass but `cl:standard-class`, and there is no way in Meta-Object Protocol that interferes in default metaclass settings without the conflict of conventional default metaclass or `cl:standard-class`. You may use `defResource` instead of `defclass` in order to define a class in the RDF universe with metaclass rdfs:Class.

Instead, if you refer any undefined class name as superclass in class definition, the undefined class is defined as forward referenced class, even in SWCLOS. Then, SWCLOS changes the class of the forward referenced class before `mop:finalize-inheritance` primary method. In other words, you must set up all classes except classes in the RDF universe by the time of the instance creation.

```
(defmethod mop:finalize-inheritance :before ((class rdfs:Class))
  (let ((forward-referenced-classes
          (remove-if-not #'excl::forward-referenced-class-p
                         (collect-superclasses* class))))
    (loop for fclass in forward-referenced-classes
       do (change-class fclass rdfs:Class))))

(defmethod mop:finalize-inheritance :before
              ((class mop:forward-referenced-class))
  (change-class class rdfs:Class))
```

Default Superclass in RDF

Rdfs8 in the rdfs rules directs every class in the RDF universe, that is an instance of rdfs:Class, to be a subclass of rdfs:Resource, and rdfs13 directs every datatype class, that is an instance of rdfs:Datatype, to be a subclass of rdf:Literal. Allegro Common Lisp is equipped with method, `excl::default-direct-superclasses`, that works in case of no superclasses defined, then we defined such methods in the RDF universe as follows. These methods are invoked when CLOS finds no superclasses in the `shared-initialize:after` method.

```
(defmethod make-instance ((class-name null) &rest initargs)
  (declare (dynamic-extent initargs))
  (apply #'make-instance 'rdfs:Resource initargs))

;; rdfs8
(defmethod excl::default-direct-superclasses ((class rdfs:Class))
  (cl:list (load-time-value (find-class '|rdfs:Resource|))))

;; rdfs13
(defmethod excl::default-direct-superclasses ((class rdfs:Datatype))
  (cl:list (load-time-value (find-class 'rdfs:Literal))))
```

Domain Constraint for Subject

Rdf4a rule describes the entailment that a subject in triple must satisfy the domain constraint of the predicate in triple. A triple in RDF corresponds to object/slot/slot-value in CLOS objects. Therefore, the domain constraint must be satisfied in the definition and redefinition of object. The following code of `make-instance` and `reinitialize-instance` illustrate the domain constraint satisfaction mechanism in SWCLOS.

```
(defmethod make-instance :around ((class rdfs:Class) &rest initargs)
  ;; make-instance (class . initargs) and make-instance (meta . initags)
  (cond ((null initargs) (call-next-method))
        (t (let ((new-domains (ensure-class-slotds class initargs)))
             (cond ((null new-domains) (call-next-method))
                   (t (let ((new-types
                             (most-specific-concepts-with-intersection
                               (cons class new-domains))))
                        (cond ((and (null (cdr new-types))
                                    (eql class (car new-types)))
                               (call-next-method))
                              ((null (cdr new-types))
                               (apply #'call-next-method
                                      (car new-types) initargs))
                              (t (warn "Multiple classing1 ~S with ~S"
                                   (getf initargs :name) new-types)
                                 (let ((shadow
                                        (make-shadow class new-types)))
                                   (apply #'call-next-method
                                          shadow initargs)))))))))))

(defmethod reinitialize-instance :around
             ((instance rdfs:Resource) &rest initargs)
  (cond ((null initargs) (call-next-method))
        (t (let ((new-domains
                  (ensure-class-slotds (class-of instance) initargs)))
             (cond ((null new-domains) (call-next-method))
                   (t (let ((new-types
                             (most-specific-concepts-with-intersection
                               (cons (class-of instance) new-domains))))
                        (cond ((and (null (cdr new-types))
                                    (eql (class-of instance)
                                         (car new-types)))
                               (call-next-method))
                              ((null (cdr new-types))
                               (apply #'change-class
                                      instance (car new-types))
                               (apply #'call-next-method
                                       instance initargs))
                              (t (warn "Multiple classing1 ~S with ~S"
                                   (getf initargs :name) new-types)
                                 (let ((shadow
                                        (make-shadow
                                          (class-of instance)
                                          new-types)))
                                   (apply #'call-next-method
                                          shadow initargs)))))))))))
```