# A Map-based Integration of Ontologies into an Object-Oriented Programming Language

Kimio Kuramitsu

**Abstract** Today's programmers have difficulties using ontology in their information-centric applications, where ontology would be useful. This paper addresses the integration technique of ontologies into an object-oriented scripting language. Our technique is based on the use of semantic mapping as a unified form of complicated semantic relations in an ontology system for the class-subclass view of an object-oriented programming modeling. This enables ordinary programmers to write ontology reasoning, such as equivalence and subsumption, without any extended logical constructors.

## 1 Introduction

The ontology technology has been widely accepted as an integral part of managing the semantics of information on the Web and other information centric systems [3]. More recently, with the popularity of the Semantic Web, practical ontology languages and tools, such as Jena [4], have been developed to share ontology through the Web. Despite of these growing concerns, there is still a huge difficulty receiving ontology benefits, especially for most of programmers who are developing web and information-rich applications where ontology would be potentially helpful.

One considerable reason is that the terminology of ontology is quit different from that of object-oriented programming languages that today's developers are very familiar with. Developers who want to use some APIs in an ontology tool, such as Jena or Fact++, have to learn about logical constructors to use the existing ontology, because they are mainly designed for KR experts to build their ontologies.

The purpose of this paper is to present a map-based approach to integrating the use of ontology into well-known constructors in an object-oriented programming language. In our approach, concepts and individuals are transparently mapped to classes and its instances, and semantic reasoning such as *equivalence* and *subsump-*

Kimio Kuramitsu
Yokohama National University, Yokohama City, Japan, e-mail: kimio@ynu.ac.jp This work has been supported in part by Grant-in-Aid for Japanese Scientific Research (1870002300) and SCOPE-R funds (062103013).

*tion* can be operated with new operators === or `isa`, which would be as friendly as `instanceof`. This enables us to write semantic program naturally like:

```
Medicine m = "Amoxillin";
if(m isa Antibiotics || m === "Penicillin")  ..
```

The strength of our map-based approach is in its ontology-language neutrality. We use *semantic mapping* as a unified view to redefine complicated conceptual relations in an ontology system. This allows us to use any type of classification-based knowledge as a part of programmed codes without external logical operators.

We will show the map-based integration through our implemented scripting language, Konoha[1]. Section 2 is an introduction of the use of ontologies in Konoha. In Section 3, we define the semantic mapping that mediates two different worlds: the ontology and the object-oriented modeling. In Section 4, we will review related work. In Section 5, we conclude the paper.

## 2 Use of Ontologies with Konoha

Every programming language has primitive types, such as int and String, which are used to represent very basic information values. However, they cannot carry any semantics that identify the concepts of its information. For example, the class String is available to represent a name of person, email, ISBN, and even an arbitrary plain text, while it provides no help for identifying the meaning of its represented string. Konoha allows us to extend primitive types, such as Int, Float, String, by adding semantic identifiers, URN (Universal Resource Name). The using statement is newly introduced to add a class to URN-specified semantic constraints.

Here is the first example, where the meaning of Celsius is added into Float. A new class, named Float::C, is generated as a result, and its instance value is associted with its semantics through the URN. (Note that, Float::C is a local name and, in global, the class is identified with URN.)

```
>>> using Float::C  http://unit/Celsius
>>> Float::C t = 20;
>>> t
20[C]
>>> t.class
Float{http://unit/Celsius}
```

Next, we suppose a vocabulary set, which is used to represent feeling temperature such as ”freezing”, ”chilly”, ”cool”, ”comfortable”.

```
>>> using String::feel  http://vocabulary/FeelTemp
>>> String::feel ft = 'feel:chilly';
>>> ft = "hello,world";       (==> InvaidValueException)
```

The class String::feel is not only semantically annotated, but also constrained in the range of its instance values. The String::feel allows to take vocabulary strings that are specified in `http://vocabulary/FeelTemp`.

[1] Our first prototyped implementation of Konoha is downloadable at http://konoha.sourceforge.jp/.

The semantic-extended class, although it is helpful for programmers to remember its meanings, is still meaningless in machine processing. That is, Konoha is able to know that Float::C and String::feel are different, but not to know whether 20C is "comfortable" or not. To obtain such a question, a reasoning system will be needed here.

In Konoha, reasoning is a part of casting/mapping between two classes. If the programmer wants to know whether 20C is comfortable, he or she can simply write as follows:

```
>>> t = 20C;
>>> (String::feel)t
"comfortable"
```

Konoha has no its own reasoning system. When it recieves a request through the mapping operation, it poses a map-based query, say, $? : -20C \mapsto \mathsf{String} :: \mathsf{feel}$ for an external ontology system, which the associated URNs indicate to. Due to the unified form of querying/answering, there is no additional library to connect the external system.

## 3 Bridging Two Worlds

### *3.1 Class and Concept*

The class, in an OO world, and the concept, in the KR world, are very similar, but they differ in that a class is specified first and its objects are instantiated after the class definition while individuals exist at first and its concept is reasoned later by classification.

As our starting point, we have chosen to build the KR concept on top of the class-first world. That is, all individuals are belonging to one existing concept from the beginning. Let $C$ be a concept name. We write $C^I$ for a set of individuals that belongs to $C$. We say $\mathtt{t} \in C^I$ if a given `t` is an instance of $C$.

Here are examples of defining two concepts AmericanSeason and BritishSeason.

$$\mathsf{AmericanSeason}^I = \{\mathtt{spring}, \mathtt{summer}, \mathtt{fall}, \mathtt{winter}\}$$
$$\mathsf{BritishSeason}^I = \{\mathtt{spring}, \mathtt{summer}, \mathtt{autumn}, \mathtt{winter}\}$$

These two concepts seem to be very similar, because both of them have the same individuals, such as `spring`, `summer`, and `winter`. However, by default, we regard these individuals as *homonyms*, i.e., the same symbols having different meanings. To identify conceptual differences between individuals, we write an instance $C.\mathtt{t}$ for $\mathtt{t} \in C^I$.

### *3.2 Semantic Mapping*

Between two concepts, there is no semantic relation by default. To add semantic relation, we use *semantic mapping*, denoted $C \mapsto D$.

To begin with, we focus on two instances $C.x$ and $D.y$. We say $C.x \mapsto D.y$ if $C.x$ is interpreted as $D.y$, the concept of $C.x$ is *broader* than that of $D.y$, or, from the perspective of relative information capacity [6], $C.x$ is more informative than $D.y$.

In addition, we say $C.x$ and $D.y$ is semantically equivalent, denoted $C.x \equiv D.y$, if and only if $C.x \mapsto D.y$ and $D.y \mapsto C.x$.

Next, we will extend the semantic mapping from two indeviduals to two concepts.

**Definition 1 (semantic mapping and equivalence).**

$$\frac{\forall x\ \exists y\ C.x \mapsto D.y}{C \mapsto D}, \quad \frac{C \mapsto D \qquad D \mapsto C}{C \equiv D} \tag{1}$$

Note that for simplicity all semantic mappings in this paper are supposed to be *total*, although *partial mappings* would be very common. In practice, we use null, the null pointer widely used in programming languages, to represent a partial mapping. We say no mapping if $C.x \mapsto \mathsf{null}$, and we write $C \not\mapsto D$ if for each $x \in C^I$ $C.x \mapsto D.\mathsf{null}$. The class $C, D$ are disjoint if $C \not\mapsto D$ and $D \not\mapsto C$.

### 3.3 Subtyping System

The *subtyping* system, generally supported in object-oriented programming languages, allows us to organize classes in a class-subclass manner. We use a partial order to represent the organized class-subclass relation; we write $C \prec D$ for the class declaration.

Konoha has the same grammar and transitivity property with Java for subtyping.

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{...\}}{C \prec D}, \quad \frac{C \prec D \qquad D \prec E}{C \prec E} \tag{2}$$

### 3.4 Bridging Ontology

An ontology is a set of *structured* terms. The "structure" is given by mathematical relations, like $\mathsf{C}(t)$ and $\mathsf{R}(t,t2)$. which are called respectively *concept* and *role*. Although different class of ontology languages [1] introduce different variation of roles, from the classification view they comonly provides three types of reasoned relations.

- (*equivalence*) $C \equiv D$,
- (*subsumption*) $C \sqsubseteq D$
- (*disjointness*) $C \sqcap D = \bot$

Note that we are interested only in these three relations due to the similarity with the class-subclass relation in object-oriented programming languages.

**Theorem 1.** *Our concept definition and semantic mapping contain $C \equiv D$, $C \sqsubseteq D$, and $C \sqcap D = \bot$.*

*Proof*(sketch). Let $\Delta$ be a finite set of terms in an ontology system. Suppose $t \in \Delta$. If a unary relation $C(t)$ is true, then we make a new instance $C.t$ in $C^I$. We always say $C.t \equiv D.t$ because $t$ is identical on $\Delta$. On the other hand, $C(t)$ is said to be true

if $C \sqsubseteq D$ and $D(t)$ is true. Accordingly, we say $C.t \mapsto D.t$ for all $t$ that satisfies both $C(t)$ and $C \sqsubseteq D$ (, i.e., $D(t)$ is true).

## 4 Related Work

There is a long history of representing knowledge representation in a LISP-style syntax. It is not unnatural to combine deductive programming features, such as Prolog, with such a LISP-style ontology description, or vice versa. More recently, Go! [2] was designed to integrate an object-oriented prolog with its own ontology description. However, the integration of a logic-based programming language with ontology constructors requires different elaborations. ActiveRDF [7] showed an ORM-style approach to the integration of RDF with Ruby, where Ruby classes are generated dynamically by SPARQL queries. This enables us to use RDF/S semantics transparently in Ruby classes. However, their mapping method is so direct that it cannot map more reasoned relations, such as equivalence and subsumption.

## 5 Conclusion

Today's programmers have difficulties using ontology in their information-centric applications, where ontology would be useful. This paper addressed the map-based integration of ontologies into an object-oriented scripting language. Our technique is based on semantic mapping, a unified form of complicated semantic relations in an ontology system for class-subclass view of an object-oriented programming modeling. Using Konoha, we showed a programmer is able to write ontology reasoning, such as equivalence and subsumption, without any extended logical constructors.

## References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider (eds). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2000.
2. Keith L. Clark and Frank G. McCabe Ontology Oriented Programming in Go! Journal of Applied Intelligence, 2005.
3. Michael Gruninger and Jintae Lee. Special issue: Ontology applications and design. *Communications of the ACM*, 45(2):39–41, 2002.
4. Jena - A Semantic Web Framework for Java. http://jena.sourceforge.net/
5. K. Kuramitsu. Mappings As A Lightweight Ontology System for the World-Wide Web. In *Proc. of the Symposium on Professional Practice in AI / IFIP World Computer Congress (WCC2004)*, 2004.
6. Renée J. Miller, Yannis E. Ioannidis, and Raghu Ramakrishnan. The use of information capacity in schema integration and translation. In *Proceedings of 19th International Conference on Very Large Data Bases*, pages 120–133. Morgan Kaufmann, 1993.
7. Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. ActiveRDF: Object-Oriented Semantic Web Programming. In *Proc. of WWW2007*, 2007.
8. Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks(eds.) OWL Web Ontology Language: Semantics and Abstract Syntax, *W3C Recommendation*, 10 February, 2004.