EVOLUTION OF ONTOLOGIES AND TYPES

Thierry Despeyroux
Inria – Paris-Rocquencourt
Domaine de Voluceau
B.P. 105
F-78153 Le Chesnay Cedex - France

ABSTRACT
Ontologies are heavily used in the context of the Semantic Web (Berners-Lee 1998, 2001) to formalize human knowledge. Ontologies engineering is now an important activity, and specialized softwares are developed to help in managing huge ontologies. The development of ontologies and of information systems can be compared to the development of programs. In this paper we make a parallel between ontologies and types in programming languages, and we use a small example to show that an ontology can be seen as a type system. When an ontology evolves, studying the impact of this evolution on the semantic annotations that use this ontology can be viewed as a type-checking process. The next step should be to import some notions used in the types community as overloading, polymorphism, type parameters, etc. to improve or create more powerful ontology definition languages.

KEYWORDS
Ontology, Type, Information system, Semantic-Web.

1. INTRODUCTION
Ontologies become more and more necessary and useful to formalize human knowledge, the size of these ontologies (in particular definitions), leads to the emergence of ontology engineering to allow the management of the creation, modification and development of huge ontologies. Some specialized software as editors are helpful (Protégé for example), and sometime database technology are required (Weithöner and al. 2004). In many aspects, the development of formal systems as ontologies is closed to the development of programs as it is already the case for web-based information systems (Pressman et al. 1998, Murugesan et al. 2001, Despeyroux 2004).

Traditional development of ontologies description formalisms refer to the world of logic (Fensel et al. 1998, Horrocks et al. 2003, Grosof et al. 2004, Patel-Schneider 2006).

In this paper we show that ontologies can be seen as type systems. Type systems have been heavily studied in the context of the semantics of programming languages (Gunter 1992). Thus it should be possible to reuse many results coming from the types community to create more powerful and more secure definition formalisms to describe human knowledge, and this is of course not incompatible with the logical vision.

Following (Luong et al. 2006, 2007) we take a small ontology and use it by means of semantics annotations, then modifying this ontology we see the effects of this evolution on the ontology on the semantics annotations. In a second step we endorse the suit of a programmer and will “implement” the same ontology as a type system, applying then the same evolution.

2. AN ONTOLOGY AND ITS EVOLUTION
An ontology is a way of representing knowledge in a formal manner. One main goal of an ontology is to be shared between a group of people to fix a terminology and the relation between concepts. Ontologies are
heavily used in semantic annotations to ease both human and machine interface and is one of the foundation of the Semantic Web.

An important word in this definition is “shared”. It implies the fact that the set of definitions in an ontology is a reference (Gruber T. R., 1993). An ontology in the context of an information system can be compared to declarations or libraries in a programming context.

As for web pages, the evolution of ontologies and of semantic annotations must be controlled. The process of evolution is studied for example in (Luong 2007, Luong et al. 2007, Luong and Dieng-Kuntz 2007).

Let’s take as example the ontology found in (Luong et al. 2007) described using RDF(S) (Klyne and Carroll 2004).

**Concepts**: Person, Trainee, PhdStudent, Manager, Researcher, Director, Team, Project;
Researcher is-a Manager;
Director is-a Manager;
Manager is-a Person;
PhdStudent is-a Person;
Trainee is-a Person;

**Properties**: work, manage;
Person work Team;
Manager manage Project;

Let’s take a list of semantic annotations, given as RDF triplets:

1. (r1 work v1) (r1 type Person)
2. (r2 work v2) (r2 type PhdStudent)
3. (r3 work v3) (r3 type Manager)
4. (r4 manage v4) (r4 type Manager)
5. (r5 work v5) (r5 type Researcher)
6. (r6 manage v6) (r6 type Researcher)
7. (r7 work v7) (r7 type Director)
8. (r8 manage v8) (r8 type Director)

In these annotations, ri and vi are instances of concepts that are inferred from the properties definitions and from type restrictions that are included in the annotations. In (Luong and al. 2007), the effects of modifying the ontology on this set of annotations are studied. The original ontology is modified, deleting the concept Director, merging PhdStudent and Trainee into the concept Student. The new ontology that is obtained is defined below:

**Concepts**: Person, Student, Researcher, Director, Team, Project;
Researcher is-a Person;
Director is-a Person;
Student is-a Person;

**Properties**: work, manage;
Person work Team;
Director manage Project;

A consequence of the modifications of the ontology is that the annotations 2, 3, 4 and 6 become inconsistent.

### 3. CONVERTING AN ONTOLOGY TO A TYPE SYSTEM
We can make a parallel between ontologies and semantic annotations with programs that are also formal systems. Concepts can be viewed as types and subsumptions as type inclusions. In this context, properties get signatures that define their domains and co-domains.

The initial ontology given in section 2. can be described as follows. The sign \( \leq \) denotes type inclusion.

\[
\begin{align*}
\text{Person, PhdStudent, Trainee, Manager, Researcher, Director, Team,} \\
\text{Project} : \text{type;}
\end{align*}
\]

\[
\begin{align*}
\text{PhdStudent} & \leq \text{Person;} \\
\text{Trainee} & \leq \text{Person;} \\
\text{Manager} & \leq \text{Person;} \\
\text{Researcher} & \leq \text{Manager;} \\
\text{Director} & \leq \text{Manager;} \\
\text{work} : \text{Person} \to \text{Team;} \\
\text{manage} : \text{Manager} \to \text{Project;}
\end{align*}
\]

The semantic annotations can be seen as expressions in a programming language. Instances are represented by objects (let’s say constants). Depending of the programming language, the type of objects can be inferred or declared. We prefer to declare them as it is the case in languages with strong typing to optimize type verification and obtain as much error messages as possible.

\[
\begin{align*}
r1 &: \text{Person;} \\
r2 &: \text{PhdStudent;} \\
r3, r4 &: \text{Manager;} \\
r5, r6 &: \text{Researcher;} \\
r7, r8 &: \text{Director;} \\
v1, v2, v3, v5, v7 &: \text{Team;} \\
v4, v6, v8 &: \text{Project;}
\end{align*}
\]

If we apply the same modifications to the ontology as previously, the type system looks now as follows:

\[
\begin{align*}
\text{Person, Student, Researcher, Director, Team, Project} : \text{type;}
\end{align*}
\]

\[
\begin{align*}
\text{Student} & \leq \text{Person;} \\
\text{Researcher} & \leq \text{Person;} \\
\text{Director} & \leq \text{Person;} \\
\text{work} : \text{Person} \to \text{Team;} \\
\text{manage} : \text{Director} \to \text{Project;}
\end{align*}
\]

Applying a traditional type checker to our set of semantic annotations, will produce error messages.

In the following declarations

\[
\begin{align*}
r2 &: \text{PhdStudent;} \\
r3, r4 &: \text{Manager;}
\end{align*}
\]

the types (concepts) PhdStudent and Manager are not declared and these declarations are not legal.

In the annotations 2, 3, and 4, \( r2, r3, \) and \( r4 \) are not of type Person as declared by the signatures of the properties work and manage. In the annotation 6, \( r6 \) is not of type Director.

(Klein 2002) shown that modifying a part of an ontology can imply some inconsistencies somewhere else in this ontology or when it is used. (Luong 2007) proposed some rules to detect these inconsistencies. Viewing an ontology as a type system, we can say that checking the consistency of ontologies and annotations can be done by a traditional type-checking.
4. CONCLUSION

In many aspects, information systems can be compared to programs. In particular, ontologies used in these information systems can be compared to the declarative parts of programs, and more precisely to type systems. In this paper, we have shown this fact on a small example, showing that checking the consistency of an ontology can be done by type-checking. Type systems have been heavily studied in the context of programming languages semantics, defining some notion as overloading, polymorphism, type parameters, etc. Modern programming languages take advantage of this in term of modularity. Ontologies definition formalisms use a lot of notion coming from the logic world. We think that a fertilization coming from the types world should be able to design more powerful and more modular ontologies definition languages.

REFERENCES

Book

Thesis

Journal

Conference paper

Technical Report

Others