Storing and Querying Ontologies in Logic Databases

Timo Weithöner[1], Thorsten Liebig[2], and Günther Specht[1]

[1] Dept. of Databases and Information Systems
University of Ulm
D-89069 Ulm
{timo.weithoener|specht}@informatik.uni-ulm.de
[2] Dept. of Artificial Intelligence
University of Ulm
D-89069 Ulm
liebig@informatik.uni-ulm.de

Abstract. The intersection of Description Logic inspired ontology languages with Logic Programs has been recently analyzed in [GHVD03]. The resulting language, called Description Logic Programs, covers RDF Schema and a notable portion of OWL Lite. However, the proposed mapping in [GHVD03] from the corresponding OWL fragment into Logic Programs has shown scalability as well as representational deficits within our experiments and analysis. In this paper we propose an alternative mapping resulting in lower computational complexity and more representational flexibility. We also present benchmarking results for both mappings with ontologies of different size and complexity.

1 Introduction

Current research within the Semantic Web aims at combining knowledge representation methods with techniques of the Web. Such a combination would enable meaningful communication between people and heterogenous information processing systems for inter- and intranet applications. Ontologies play a pivotal role within such a framework by providing a shared and common understanding of a domain of interest. Formally, an ontology is a logical theory accounting for the intended meaning of a formal vocabulary, i. e. its ontological commitment to a particular conceptualization of the world [Gua98].

Reasoning about logical theories requires logic-based inference systems which has been well studied within the field of knowledge representation in the AI community over the last decades. Description Logics (DL's) as a decidable fragment of first-order logic turned out to be an adequate formalism for representing and reasoning about expressive ontologies. As a consequence DL's form the formal foundation of W3C's Web Ontology Language (OWL), a proposed standard for a semantic markup language for publishing and sharing ontologies on the World Wide Web.

One of the key design goals for OWL was highest possible expressiveness [Hef03]. However, the more expressive a language is, the more difficult it will be to learn or to use this language. OWL has been criticized because of its high language complexity even by one of its language designers [vH02]. While analyzing online accessible ontologies the same language designer also noticed, that even experienced users exploit a very limited subset of the available language primitives in general. Beyond that, reasoning about ontologies with an expressiveness comparable to that of OWL requires sophisticated logical theorem provers, which are currently only available as research prototypes. Such systems have proven to be fast as well as reliable at least with most of the academic ontologies available so far. However, they are designed to deal with ontologies completely processable within a computers primary memory. In fact, we expect much larger ontologies for real world applications in the near future which very likely will not solely be loadable into (virtual) main memory. More concrete, realistic applications scenarios within the vision of the Semantic Web refer to (currently non-existent) ontologies with a limited set of language primitives but a very large set of individuals (flight schedules, phone books, etc.). Obviously, database technology will be necessary in order to be able to deal with ontologies of this size.

In this sense, as an alternative to tableaux-based DL theorem provers Grosof et. al. [GHVD03] recently suggested a mapping of a DL subset into Logic Programs (LP) suitable for evaluation with Prolog. This intersection of DL with LP called DLP completely covers RDF Schema and a fraction of OWL (notably most of OWL Lite extended with general concept inclusion). This approach is called the "Direct Mapping" approach in the following. Logical database systems seem most suitable to combine LP with efficient and persistent data storage. However, applying the Direct Mapping approach for loading, storing and evaluating ontologies in logical database systems has shown some significant scalability deficits as well as representational drawbacks. Therfore, we developed a new mapping without this limitations which we call the "Meta Mapping" approach below. This approach is meta in the sense that it maps the LP subset of OWL into a higher representational level resulting in lower computational complexity and more representational flexibility. For this reason the Meta Mapping approach is especially suitable for storing and processing ontologies within logical databases. In this paper we present our new Meta Approach together with some benchmarking results for both approaches.

The remainder of this paper is organized as follows. In the next section we will give an overview over logic based ontology languages currently proposed by the W3C and their relationship to Logic Programs as used in logic databases. Readers familiar with OWL and Logic Programs should skip Section 2. Section 3 explains the Direct Mapping approach from the DLP fragment of OWL to LP's proposed by Grosof and Horrocks et. al. [GHVD03]. In Section 4 we present our Meta Mapping approach while making use of examples showing the conceptual differences between the two approaches. Section 5 contains benchmarking results for both mappings with ontologies of different size and complexity. We will end with a discussion about the pros and cons of the different mappings.

2 Preliminaries

This section will shortly introduce OWL. In particular, we will give syntax examples and their semantics in terms of corresponding First Order Logic (FOL) formulae. We then characterize the intersection of LP's with OWL. Reader familiar with OWL and LP's may skip this section.

2.1 Ontology Languages for the Web

The proposed mechanism for meaningful communication between people and / or machines within the World Wide Web is to add semantic markup to Web resources in order to explicitly describe their content. This semantic markup makes use of terms for which ontologies provide a concrete specification of their meaning.

The significant term structure of ontology languages currently under development for the Web consists of at least two elements (see also [Hef03]): *classes* and *relationships* (called properties) that can exist among classes.

RDF Schema. The two basic structuring elements from above are provided by the Resource Description Framework Schema (RDFS), the lowermost ontology language of the Semantic Web language layer architecture. As with the following ontology languages, RDFS usually is serialized as XML document in order to meet the syntactical requirements of todays Web communication protocols. RDFS can be considered as a very simple ontology language allowing the definition of class hierarchies via `subClassOf` statements. Exemplarily, a dog can be defined as some kind of mammal as follows:

```
<rdfs:Class rdf:ID="Dog">                                    [Ex. 1]
  <rdfs:subClassOf rdf:resource="#Mammal"/>
</rdfs:Class>
```

Semantically this can be expressed in FOL as an implication between two unary predicates: $\forall x : \texttt{Dog}(x) \Rightarrow \texttt{Mammal}(x)$ (DL abstract notation: $\texttt{Dog} \sqsubseteq \texttt{Mammal}$).

The possible combinations of classes and properties can be restricted by qualifying the domain and range of properties. An owner relationship for dogs, called `Dog-Owner`, is narrowed in its domain to the class `Human` and in its range to `Dog` in the following:

```
<rdf:Property rdf:ID="Dog-Owner">                            [Ex. 2]
  <rdfs:domain rdf:resource="#Human"/>
  <rdfs:range rdf:resource="#Dog"/>
</rdf:Property>
```

The FOL correspondence to properties are binary predicates. According to that, the semantics of the property definition in example 2 is as follows: $\forall x, y : \texttt{Dog-Owner}(x, y) \Rightarrow \texttt{Human}(x)$ and $\forall x, y : \texttt{Dog-Owner}(x, y) \Rightarrow \texttt{Dog}(y)$ (DL: $\top \sqsubseteq \forall \texttt{Dog-Owner}.\texttt{Human}$, $\top \sqsubseteq \forall \texttt{Dog-Owner}^{-}.\texttt{Dog}$).

Property hierarchies can also be defined analogous to class hierarchies using `subPropertyOf` statements.

Web Ontology Language (OWL). OWL is developed as a vocabulary extension of RDF Schema[1] and is derived from the DAML+OIL Web ontology language. This extension covers class language constructs like conjunction, disjunction, negation, existential and universal qualified quantification and cardinality constraints of properties (plus some others). OWL itself provides three increasingly expressive sublanguages. The least expressive sublanguage is OWL Lite. With focus on the intersection of OWL with Logic Programs we will give a more detailed explanation for some of OWL Lite's language constructs here. For syntax and semantics of all constructs see [vHHH+03] resp. [PSHH03].

In OWL a class can be defined as conjunction of other classes or class descriptions using the `intersectionOf` statement. For example, it might be rational to define `Puppy` as the conjunction of the classes `Dog` and `Young-Animal`:[2]

[Ex. 3]
```
<owl:Class rdf:ID="Puppy">
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Dog"/>
        <owl:Class rdf:about="#Young-Animal"/>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
```

Semantically, this corresponds to logical conjunction in FOL: $\forall x : \texttt{Puppy}(x) \Rightarrow \texttt{Dog}(x) \wedge \texttt{Young-Animal}(x)$ (DL: $\texttt{Puppy} \sqsubseteq \texttt{Dog} \sqcap \texttt{Young-Animal}$).

So far, all class definitions result in logical implication with respect to their definition. They are therefore called necessary class definitions. In contrast, it is possible to give a necessary as well as sufficient definition for a class. In the following example it is a necessary as well as sufficient condition being a `Dog` and a `Rabit-Animal` for being a `Rabit-Dog`:

[Ex. 4]
```
<owl:Class rdf:ID="Rabit-Dog">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Dog"/>
    <owl:Class rdf:about="#Rabit-Animal"/>
  </owl:intersectionOf>
</owl:Class>
```

Logically this corresponds to an equivalence between `Rabit-Dog` and its defining conjunction: $\forall x : \texttt{Rabit-Dog}(x) \Leftrightarrow \texttt{Dog}(x) \wedge \texttt{Rabit-Animal}(x)$ (DL: $\texttt{Rabit-Dog} \equiv \texttt{Dog} \sqcap \texttt{Rabit-Animal}$).

Universal qualified quantification is a language construct for locally restricting the range of a given property within a class definition. E. g., a `Doghouse` is a `house` for which all fillers of the property `Occupants` are of type `Dog`:

[1] However, OWL does not include RDFS's recursive meta model property.

[2] Since OWL is layered on top of RDFS the examples from above are easily converted into OWL by changing `rdfs:Class` into `owl:Class` and `rdf:Property` into `owl:ObjectProperty`.

```
<owl:Class rdf:ID="Doghouse">                                    [Ex. 5]
  <rdfs:subClassOf>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#House"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#Occupant"/>
          <owl:allValuesFrom rdf:resource="#Dog"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
```

The fragment from above has the following semantics in FOL: $\forall x : \texttt{Doghouse}(x) \Rightarrow \texttt{House}(x) \wedge (\forall y : \texttt{Occupants}(x, y) \Rightarrow \texttt{Dog}(y))$ (DL: $\texttt{Doghouse} \sqsubseteq \texttt{House} \sqcap \forall \texttt{Occupants}.\texttt{Dog}$).

Additionally, OWL Lite extends RDFS for transitive, inverse, and symmetric properties. A perfect example of a transitive property within our dogs world is the descendants relationship `Dog-Offspring`:

```
<owl:TransitiveProperty rdf:ID="Dog-Offspring"/>                 [Ex. 6]
```

Semantically, a transitive property enforce that: $\forall x, y, z : (\texttt{Dog-Offspring}(x, y) \wedge \texttt{Dog-Offspring}(y, z)) \Rightarrow \texttt{Dog-Offspring}(x, z)$ (DL: $\texttt{Dog-Offspring}^{+}$). An inverse property of a given property can be defined as follows:

```
<owl:ObjectProperty rdf:ID="Is-Dog-Of">                          [Ex. 7]
  <owl:inverseOf rdf:resource="#Dog-Owner"/>
</owl:ObjectProperty>
```

In FOL this means: $\forall x, y : \texttt{Is-Dog-Of}(x, y) \Rightarrow \texttt{Dog-Owner}(y, x)$ and vice versa (DL: $\texttt{Is-Dog-Of} \equiv \texttt{Dog-Owner}^{-}$). A symmetric property can be introduced as follows:

```
<owl:SymmetricProperty rdf:ID="Friend-Of"/>                      [Ex. 8]
```

This definition imposes the following FOL semantics: $\forall x, y : \texttt{Friend-Of}(x, y) \Rightarrow \texttt{Friend-Of}(y, x)$.

Until now, we have only defined a formal vocabulary. In DL terminology such a vocabulary is called a TBox. In contrast, an ABox specifies concrete individuals with respect to a given TBox vocabulary. For example, `Fluffy` as a concrete dog can be defined by instantiating the class `Dog`:

```
<Dog rdf:ID="Fluffy"/>                                           [Ex. 9]
```

Such a class instantiation corresponds to an unary predicate instantiation in FOL (and abstract DL): $\texttt{Dog}(\texttt{Fluffy})$. Let us assume we also want to assert a particular doghouse (with name `Fidos-Kennel`) occupying `Fido` an animal with rabies. FOL equivalent: $\texttt{Doghouse}(\texttt{Fidos-Kennel}), \texttt{Occupant}(\texttt{Fidos-Kennel}, \texttt{Fido})$ and $\texttt{Rabit-Animal}(\texttt{Fido})$:

```
<Doghouse rdf:ID="Fidos-Kennel">                                    [Ex. 10]
  <Occupant>
    <Rabit-Animal rdf:ID="Fido"/>
  </Occupant>
</Doghouse>
```

The underlying semantics of our definitions allows to infer logically entailed knowledge implicitly encoded in our dogs world ontology. Usually, logical reasoning systems are used for making such knowledge explicitly available for users.

Concerning our example a simple inference chain could be the following. Since all occupants of a doghouse necessarily have to be dogs (due to the definition of `Doghouse` in Example 5) it follows, that `Fido` has to an instance of class `Dog`. As a consequence `Fido` is also classified as a `Rabit-Dog` because being a dog and a rabid animal is sufficient for being a `Rabit-Dog`.

2.2 Intersecting OWL with Logic Databases

Based on the analysis of the intersecting language of DL with LP in [GHVD03] we will shortly characterize the resulting language of the intersection of DL inspired OWL with LP in the following.

Logic Programs (LP) consist of a set of rules having the form:

$$A \leftarrow B_1 \wedge \ldots \wedge B_n \qquad \text{with} \quad n \geq 0$$

where A, B_i are atomic formula of predicates of arbitrary arity. A is called the head of the rule and the conjunction of B_i's is called the body. Atomic formula are allowed to be "negated". Note that this negation has to be interpreted as negation-as-failure. Negation-as-failure as well as other LP features (like procedural attachments) are not expressable in FOL and therefore also not expressable in logic based ontology languages. On the other hand predicates in LP are not restricted in their arity in contrast to DL. In addition DL restricts the usage of (implicit) free variables in quantifying expressions with guarding property predicates. As a result the intersection of DL with LP can be characterized as a definite (without negation-as-failure) equality-free Horn fragment of FOL, called Description Logic Programs (DLP) in [GHVD03].

Since OWL is a DL inspired language it covers DLP completely. More precise, DLP covers most of OWL Lite (the least expressive sublanguage of the OWL family) plus a portion of OWL DL, namely general concept inclusions (GCIs) with disjunction and qualified existential qualification on the l.h.s. However, GCIs are not very common in ontologies so far, we will focus on the intersection of OWL Lite with LP, which we will call OWLP Lite for the rest of the paper.

3 The Direct Mapping Approach

In the following we will provide two approaches of converting ontologies into logic programs. Starting with the previously published proposal in [GHVD03], the

direct mapping approach, we will suggest an alternative meta mapping approach in Section 4. After that we will provide an evaluation and comparison of both approaches, which makes it necessary to have a look at different aspects like the number of facts and rules contained in the resulting logic programs.

A straight forward approach to convert ontologies into a logic program is described in [GHVD03]. This approach maps every class or property definition contained in the ontology into a rule and every class-instance or instance-property-instance relationship into a fact. In the following we summarize this method of mapping into a description logic program.

3.1 The Mapping

Every concept instantiation is mapped to a unary relation with the concept name becoming the name of the relation and the individual name becoming the only argument. For example the statement that `Fluffy` is an instance of concept `Dog` (see Example 9) is mapped into the fact:

```
Dog("Fluffy").
```
[Ex. 11]

Every instance-property-instance relationship is mapped into a binary relation with the property's name becoming the name of the relation. The first argument is the name of the individual, the second argument is the property's value. Given a property `Occupant`, let us assume `Fidos-Kennel` being occupied by `Fido` (as defined in Example 10):

```
Occupant("Fidos-Kennel", "Fido").
```
[Ex. 12]

In addition, concept as well as property constructor statements are converted into a set of rules. In this step it gets obvious, why we called this approach the Direct Mapping approach. Every subclass relationship stated in the ontology is directly mapped into a corresponding logic program rule. As a consequence the OWL fragment defined in Example 1 is converted into:

```
Mammal(X) :- Dog(X).
```
[Ex. 13]

For a mapping of the rest of the OWLP Lite language constructs see Table 1. Note, that the EquivalentClasses relationship is just a mutual SubClassOf relationship. An EquivalentClasses relationship can be simulated by two SubClassOf statements using both defintion directions.

3.2 Size of the Resulting Program

Let us now have a look at the resulting logic program of a given ontology with a set $\mathcal{C}$ of classes and a set $\mathcal{P}$ of different properties. Let $\mathcal{R}_{\mathtt{c}}$ with $\mathtt{c} \in \mathcal{C}$ be the set of rules required to express `c`. With $\mathcal{I}$ being the set of class instantiations (see Example 11) and $\mathcal{V}$ the set of property instantiations (see Example 12) defined within the ontology we get an total of

Table 1. Shows OWL statements and there representation in a logic program as suggested [GHVD03]. See Section 3.2 for definition of $\mathcal{R}_{\mathtt{c}}$.

OWL Abstract Syntax [PSHH03] Definition of class c	DLP Statements Rule set $\mathcal{R}_{\mathtt{c}}$	$\lvert\mathcal{R}_{\mathtt{c}}\rvert$
SubClassOf(c b)	b(X) :– c(X).	1
SubClassOf(unionOf($\mathtt{b}_1$ ... $\mathtt{b}_n$) c)	c(X) :– $\mathtt{b}_1$(X). ... c(X) :– $\mathtt{b}_n$(X).	n
SubClassOf(c intersectionOf($\mathtt{b}_1$ ... $\mathtt{b}_n$))	c(X) :– $\mathtt{b}_1$(X), ..., $\mathtt{b}_n$(X).	1
SubClassOf(intersectionOf($\mathtt{b}_1$ ... $\mathtt{b}_n$) c)	$\mathtt{b}_1$(X) :– c(X). ... $\mathtt{b}_n$(X) :– c(X).	n
SubClassOf(c restriction(p allValuesFrom(b)))	c(X) :– p(X, b), anonID(X).	1
SubClassOf(restriction(p someValuesFrom(b)) c)	anonID(X) :– p(X, b), c(X).	1

$$\sum_{\mathtt{c}\in\mathcal{C}} |\mathcal{R}_{\mathtt{c}}| \tag{1}$$

rules and

$$S = |\mathcal{I}| + |\mathcal{V}| \tag{2}$$

facts, while the number of different predicates is

$$K = |\mathcal{C}| + |\mathcal{P}| \tag{3}$$

To understand of the above we will establish two criteria to compare different ontologies: Size and complexity of an ontology. Size means the number of concept and property instantiations included (see S in Equation 2) while complexity means the number of different concepts (see K in Equation 3). For example you would expect that an ontology made from data contained within a phonebook, would be of very limited complexity (containing only a very limited number of concepts like name, address and phone number) but of huge size (listing all inhabitants and thus having lots of individuals). Nevertheless the number of rules in the resulting logic program is growing linear with the number of concept and property definitions in the ontology.

3.3 A first assessment of the approach

Limitations. Looking at the logic program fragments as discussed above we soon realize that this approach has some significant weaknesses:

– First, the concept names cannot be accessed from within the logic program. For that reason it is virtually impossible to get an answer to the question "give me all classes the individual I is instance of".

Reconsider Example 10 out of our dogs ontology. We are aware of the fact that `Fido` is a `Rabit-Animal`. However we are not able to retrieve all classes `Fido` is an instance of unless we *manually* walk through every possible class and ask whether `Fido` is instance of this class. Which is not very efficient and would require to know all class names of the ontology.
- The transformation creates a limited number of facts. One for every class instantiation and one for every property instantiation. The number of different relations depends on the number of classes and properties defined in the ontology. One can expect to get a very limited number of facts per relation. While the number of facts remains manageable the number of different rules grows linear with the complexity of the ontology (the knowledge included within). See Table 1 for the number of rules needed to express a given OWL statement.
- The names of the relations used and the structure of the rules involved vary from ontology to ontology. Consequently precompilation or query optimization become an real issue in this approach.

Possible Improvements. Two things have to be done to overcome the limitations mentioned above.

1. The rules and facts have to be pushed to a meta level, where names of concepts and properties become arguments of "meta predicates". As a result concept and property names can be reached from within the logic program easily.
2. We should try to get a constant set of rules valid for all ontologies accompanied by a set of fact predicates with constant names. Like this queries would look the same for every single ontology with the only difference in the amount of facts that have to be processed to get an answer.

The following section will suggest an approach which will produce a logic program following the above considerations.

4 The Meta Mapping Approach

This section will describe our approach of mapping an ontology into a logic program suitable for a deductive database. We will show that even if an ontology grows in complexity the resulting logic program will have a constant number of rules and predicates. And we will show how ABox as well as TBox reasoning become possible without the limitations from above.

4.1 The Basic Idea

Basically we convert the OWL statements contained in an ontology into a set of facts reflecting the content of the ontology. Coming from the Direct Mapping approach we basically push all facts defined there to a meta level comparable (at

least in parts) to the HiLog [CKW93] approach which has a higher-order syntax and allows terms to appear in places where predicates and atomic formulas occur in FOL. The meta mapping is realized by two new relations, which collect all facts defined in the various relations of the Direct Mapping approach.

Class Instantiation. Asserting instance `i` to be of class `C` results in instantiating the binary relation named `type` in the following way:

```
type("i", "C").
```

Property Instantiation. Likewise property instantiations are mapped into a relation named `propInst` with three arguments and constant name `propInst`. Arguments are the property name `P`, instance name `i` and property value `v`:

```
propInst("P", "i", "v").
```

Compared to the "Direct Mapping" Approach we avoid having one additional relation per property definition, by pushing the relation names into predefined "meta relations". As a consequence concept and property names are now easily accessible from within the logic program. Table 2 compares the resulting logic program fragments from Examples 9 and 10 for both approaches.

Table 2. Individual and property definition in the different approaches

	Direct Approach suggested in [GHVD03]	Our Meta Approach
`Fluffy` is a `Dog` `Fido` is the `Occupant` of `Fidos-Kennel`	`Dog("Fluffy").` `Occupant("Fidos-Kennel", "Fido")`	type("Fluffy", "Dog"). propInst("Occupant", "Fidos-Kennel", "Fido").

Handling of Class Constructors. Let us once again have a look at the very basic OWL fragment stating that concept `Dog` is a subclass of concept `Mammal` as defined in Example 1. The Direct Mapping approach would create a rule which would say that every individual of type `Dog` is also an individual of type `Mammal` (See Example 13). Please note that the explicit knowledge of the subclass relationship gets lost. All we still know is that every `Dog` is also a `Mammal`. In the Meta Mapping approach subclass relationships or any other kind of constructors are not converted into a rule covering the meaning of the statement but into a fact which states, that the ontology defines such a relationship. Consequently the number of rules is not increased if a new class is constructed and no new relations are needed, as there is a fixed number of predefined relations, reflecting the vocabulary of OWLP Lite. The above example would consequently look like this in the Meta Mapping approach:

```
isSub("Dog", "Mammal").
```
[Ex. 14]

In order to reflect the underlying semantic of the introduced meta relations, we will have to add some rules, which work on the given facts. The following rule defines that if an individual `I` is instance of concept `Y` and `Y` is subclass of concept `X`, `I` is also an individual of class `X`:

```
type(I, X) :- isSub(Y, X), type(I, Y).
```

As the rule can be used with any combination of bound and free variables, every kind of class-instance query (ABox reasoning) is possible (in contrast to the Direct Mapping approach). Additionally the transitivity of the subclass relationship is covered by the following rule:

```
isSub(X, Y) :- isSub(X, Z), isSub(Z, Y).
```

As you can see the above rules are completely independent of any entities defined in the ontology and can thus be used for every ontology. With the combination of the ontology specific facts and the general rule we can now perform all A- and TBox queries.

Table 3. Shows how A- and TBox reasoning is performed in the different approaches

Query	Meta Approach	Direct Approach
Is given individual `i` instance of given class `C`?	`?type("i", "C").`	`?C("i").`
List all instances of given class `C`.	`?type(I, "C").`	`?C(I).`
List all classes given individual `i` is instance of.	`?type("i", C).`	Manually go through every known class `C` and check for `?C("i").`
Check if given class `C` is subclass of given class `D`.	`?isSub("C", "D").`	Create new instance $\texttt{i}_\texttt{C}$ of class `C` and check whether $\texttt{i}_\texttt{C}$ is also instance of class `D`.
List subclasses of given class `C`.	`?isSub(X, "C").`	Manually go through every known class `D` create new instance $\texttt{i}_\texttt{D}$ from this class. Check whether $\texttt{i}_\texttt{D}$ is also instance of `C`.

While the queries in our Meta Mapping approach only require basic knowledge of the entities defined in the ontology, Direct Mapping approach requires complete knowledge of all class names defined to be able to perform any class hierarchy query. This constitutes a big disadvantage for the user, as he either has to keep track of all class names and the results of his queries are always questionable or he has to provide and use additional predicates providing class (and property) names.

4.2 The Rule Set

In the above section we mentioned two rules which provide the logic behind the meta level relations we defined to store the ontologies knowledge. Depending on the number of different constructors used in the ontology this set of rules will vary in size. But the size of this rule set is not depending on the size or complexity of the ontology. E.g. if an ontology uses the `intersectionOf` statement the corresponding rules have to be added to the logic program. Any further use of the `intersectionOf` statement will not increase the rule set. Nevertheless for the following considerations we assume the rule set to be constant. This is achieved by working with the complete rule set, containing rules for every OWLP Lite statement no matter if they are used in the ontology or not. Table 4 shows some of the required facts for comparision with the Direct approach (see Table 1). Table 5 gives a nearly complete summary of the required rules and facts of our Meta Mapping approach.

Table 4. Shows OWL statements and there representation in a logic program

OWL Abstract Syntax Definition of class c	DLP Statements Fact set $\mathcal{F}_{\mathtt{c}}$	$\lvert\mathcal{F}_{\mathtt{c}}\rvert$
SubClassOf(c b)	isSub(c, b).	1
SubClassOf(c unionOf($\mathtt{b}_1$ ... $\mathtt{b}_n$))	rhsUnionOf(c, {$\mathtt{b}_1$,..., $\mathtt{b}_n$}).	1
SubClassOf(c intersectionOf($\mathtt{b}_1$ ... $\mathtt{b}_n$))	rhsIntersectionOf(c, {$\mathtt{b}_1$,..., $\mathtt{b}_n$}).	1
SubClassOf(intersectionOf($\mathtt{b}_1$ ... $\mathtt{b}_n$) c)	lhsIntersectionOf(c, {$\mathtt{b}_1, ..., \mathtt{b}_n$}).	1
SubClassOf(c restriction(p allValuesFrom(b)))	rhsAllValuesFrom(c, p, b).	1
SubClassOf(restriction(p someValuesFrom(b)) c)	lhsSomeValuesFrom(c, p, b).	1

4.3 Influence of the Ontology's Size

Again we have to consider the size of the resulting logic program in dependence of the size and complexity of the given ontology. With $\mathcal{I}$ being the set of individuals, $\mathcal{P}$ the set of properties defined in the ontology, $\mathcal{V}$ the property values defined, $\mathcal{C}$ being set of concepts defined within the ontology and finally $\mathcal{F}_{\mathtt{c}}$ with $\mathtt{c} \in \mathcal{C}$ being the set of facts required to express concept c we get an total of

$$|\mathcal{I}| + |\mathcal{V}| + \sum_{\mathtt{c} \in \mathcal{C}} |\mathcal{F}_{\mathtt{C}}| \tag{4}$$

facts while the number of different predicates as well as the number of different rules remains constant independent of the ontology. Depending on the implementation – and the optimizations within – these numbers may vary but

Table 5. Shows a selection of the rules required in the Meta Mapping approach.

DL syntax	"Meta Mapping" representation

$i : C$	`type("i", "C").`
	`type(I, C) :- type(I, Z), isSub(Z, C).`
	If individual I is instance of Z and Z is subclass of class C, I is also instance of class C.

$(i, v) : P$	`propInst("P", "i", "v").`
	`propInst(P, I, V) :- propInst(Q, I, V),`
	`subPropertyOf(Q, P).`
	If an instantiation I,V holds for property Q, it also holds for any property P that Q is subproperty of.

$C \sqsubseteq D$	`isSub("c", "d").`
	`isSub(C, D) :- isSub(C, Z), isSub(Z, D).`
	The subclass relationship is transitive.

$P \sqsubseteq Q$	`subPropertyOf("P", "Q").`
	`subPropertyOf(P, Q) :- subPropertyOf(P, Z),`
	`subPropertyOf(Z, Q).`
	The subproperty relationship is transitive.

$\top \sqsubseteq \forall P.C$	`domain("P", "C").`
	`type(I, C) :- propInst(P, I, V), domain(P, C).`
	If instance `I` is in the domain of a relation `P` with a domain restriction on class `C`, then `I` is an instance of `C`.

$\top \sqsubseteq \forall P^{-}.C$	`range("P", "C").`
	`type(V, C) :- propInst(P, I, V), range(P, C).`
	If instance `V` is in the range of a relation `P` with a range restriction on class `C`, then `V` is an instance of `C`.

$C \sqsubseteq M_1 \sqcap \cdots \sqcap M_n$	`rhsIntersectionOf("C", {"M1", ..., "Mn"}).`
	`isSub(C, M) :- rhsIntersection(C, S), member(S, M).`
	Class `C` is subclass of any of the members of the intersection.

$M_1 \sqcap \cdots \sqcap M_n \sqsubseteq C$	`lhsIntersectionOf("C", {"M1", ..., "Mn"}).`
	`oneOfOtherType(I,S) :- member(S,M), not type(I,M).`
	`type(I, C) :- lhsIntersectionOf(C, S),`
	`not oneOfOtherType(I, S).`
	An Individual `I` is an instance of `C` if there is no member of the intersection of which `I` is not an instance of.

$C \sqsubseteq \forall P.D$	`rhsAllValuesFrom("C", "P", "D").`
	`type(I, D) :- rhsAllValuesFrom(C, P, D),`
	`type(X, C), propInst(P, X, I).`
	If instance `X` of class `C` is related to an individual `I` via a property `P` and the range of `P` is locally restricted to `D` in `C`, then `I` must be of type `D`.

$P^{+} \sqsubseteq P$	`transitiveProp("P").`
	`propInst(P, I, V) :- transitiveProp(P),`
	`propInst(P, I, Z), propInst(P, Z, V).`
	If `P` is a transitive property then for all property instantiations `I,Z` and `Z,V` also the instantiation `I,V` holds.

either way, we are talking about a rule set, which should not exceed 20-40 rules and about the same quantity of different predicates.

More precisely linear increase of ontology size and/or complexity leads to a linear growth of facts but a constant set of relations and rules. In contrast applying the Direct Mapping approach would result in linear growth of relations, rules and facts. The following section provides a detailed performance analysis with respect to the different approaches in a logical database.

5 Evaluation

In the preceding sections of this paper we discussed two different approaches for mapping ontologies into description logic programs. While introducing the approaches, we came across a number of conceptual issues, which in some cases resulted in a loss of functionality. In other cases we presumed influence on the preformance of the resulting logic program. Especially as ontologies are expected to rise in size (and complexity), we will have to think about reasoning systems, which are not entirely depend on main memory (like todays tableau reasoners or prolog) but are able to work on data in secondary storage structures, which immediately leads to database technology.

5.1 The Selected Database System

Deductive Database Systems. Due to the representation of the ontologies as logic programs it is obvious to chose a logic database or – to be more specific – a deductive database to store and query the knowledge contained in the ontologies. In the years 1988-1992 a number of deductive database systems like LDL, NAIL, LOLA and CORAL have been developed [Spe91]. Novel features of this systems have been:

1. Logic programs similar to Prolog replaced SQL as database definition and query language.
2. Arbitrary recursion (left/right hand side, quadratic) where introduced.
3. Deductive Databases abolished NF1 and allowed arbitrary (even recursiv) term structures.

Basics of logic databases are a set at-a-time (not tuple at-a-time) bottom-up (not top-down) evaluation. To do so logic programs are converted into a relational algebra program internally, where each predicate symbol is represented by a relation, rules correspond to views and each fact can be understood as an entry in a base relation. Additionally in the rule body joins are created from conjunctions over predicates with equal variable names and instantiated terms are converted into selections. Save negations become antisemijoins (a special kind of set difference) and recursion is evaluated using delta iteration or seminaive iteration respectively. Finally the transition from rule body to head becomes a projection.

Some of these features, particularily recursion and abolishment of NF1, have been re-adopted by the relational database technology. Consequently you will find some of the above in SQL:99, but there are still limitations like quadratic recursion (which is not used in either the Meta-[3] nor the Direct Mapping approach).

Even though in the future a further mapping of the logic programs into SQL:2003 might be desirable, we base the following considerations upon a deductive database and thus stay with the logic programs as described in Sections 3 and 4. As both – relational and deductive – systems work with relational algebra programs internally, we assume only minor influence on the following discussion.

The Deductive Database System Coral. For the implementation of our test cases we have chosen the deductive database system CORAL [RSSS94], as it is a stable, well tested and easily available system. Special technical features of other deductive databases that would qualitatively lead to different results will also be addressed in the following.

5.2 Measurements

When measuring the performance of a deductive database processing a logic program two measures have to be considered. First we need to have a look at the time needed to read the logic program into the database and second we have to measure the time it takes to run the logic program (perform a given query). There are certain optimization steps like magic set transformation, which are performed at different times depending on the database system used. E.g. the deductive database system LOLA [FSS91] is performing this optimization task after the query is known, only considering those rules required for the specific query. In contrast the CORAL system is performing the magic set transformation (and other optimization like supplementary magic) when the logic program is initially loaded into the database. Thus *all* binding combinations[4] are precompiled in CORAL. This gives us the opportunity to measure the time needed to perform this optimization tasks independetly from the time needed to perform certain queries.

5.3 Testcase

We used a set of different ontologies for our performance tests. Even though these ontologies vary in size and complexity our main focus was the influence of the ontologies size.

[3] In fact you will find an example of a quadratic recursion in Section 4.1. This recursion can easily be converted into a left/right hand side recursion in an optimized implementation.

[4] arguments in the query may be bound (without variables) or free (variables or terms containing variables).

Testcase 1. The first set of ontologies, consisted of the a class hierarchy of 20 classes with a varying number of individuals. We used this test case to show the influence of the ontology's size. We measured the time needed for preprocessing and time needed to process a class instance query.

Testcase 2. This test case consisted of a set of ontologies with a growing number of classes and a constant number of one instance per class. It was used to show the influence of the complexity of an ontology. We measured the time needed for preprocessing and one class subsumption query.

5.4 Test Environment

All tests were performed on a Pentium II (350 MHz) Linux system (kernel 2.4.19) with 250MB main memory running CORAL Version 1.5.2. Time Measurement was performed by the CORAL builtin commands `reset_timer().` and `display_timer().`, which where included in the logic programs. We present "CPU Time" measurements in this paper.

5.5 Results

Loading and Preprocessing the Logic Program. Figures 2 and 1 show that, our Meta Mapping approach outperformed the Direct Mapping approch when preprocessing the logic program. In fact preprocessing time grows linear with the number of instances and classes defined in the ontology. In contrast to the Direct Mapping approach, showing exponential rise (see Figure 2) of preprocessing time with a growing number of classes. In our test environment we very easily reached a point where CORAL was not able to load such a logic program within reasonable time[5] while loading the same ontology converted into a Meta Mapping approach logic program took only seconds. This also meant that we were unable to perform a comparison of query time for really huge ontologies. Only the Meta Mapping approach is able to handle this case.

Querying the Logic Program. When comparing query times you have to keep in mind a number of issues which we will highlight in the following paragraphs.

Some queries can't be performed in the Direct Mapping approach. For example you can't get a list of all concepts defined in an ontology. Thus there is nothing to compare with the Meta Mapping approach.

Additionally there are a number of queries which cannot be performed in the logic program itself when using the Direct Mapping approach. Some queries would require some kind of manual interaction or scripting which – again – makes it useless to compare the runtime of queries.

CORAL as well as other deductive database systems work as described in Section 5.1 but in general do not use secondary storage structures, by now. It is

[5] We aborted all tests after 30 minutes.

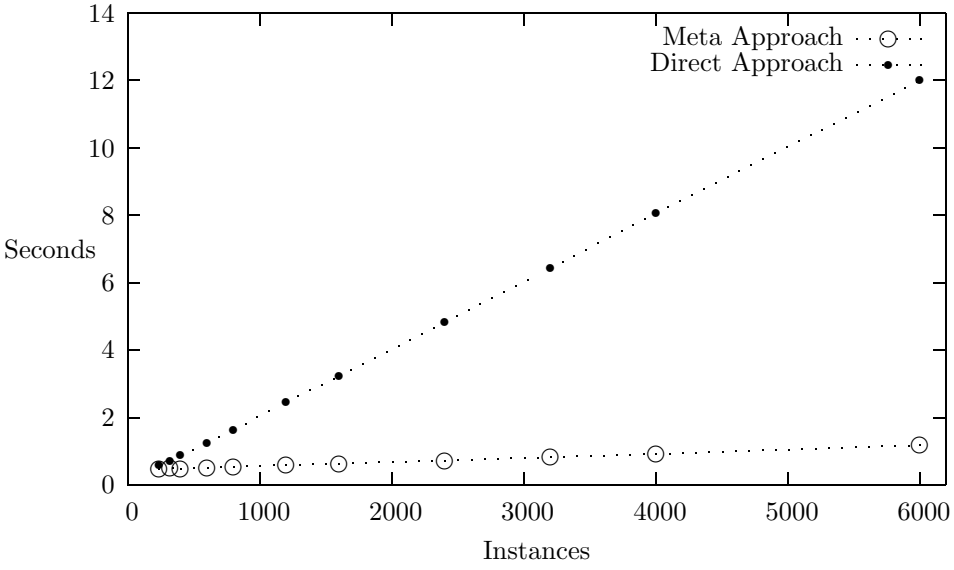

Fig. 1. Shows the time needed to preprocess the ontologies of Testcase 1 (see Section 5.3).

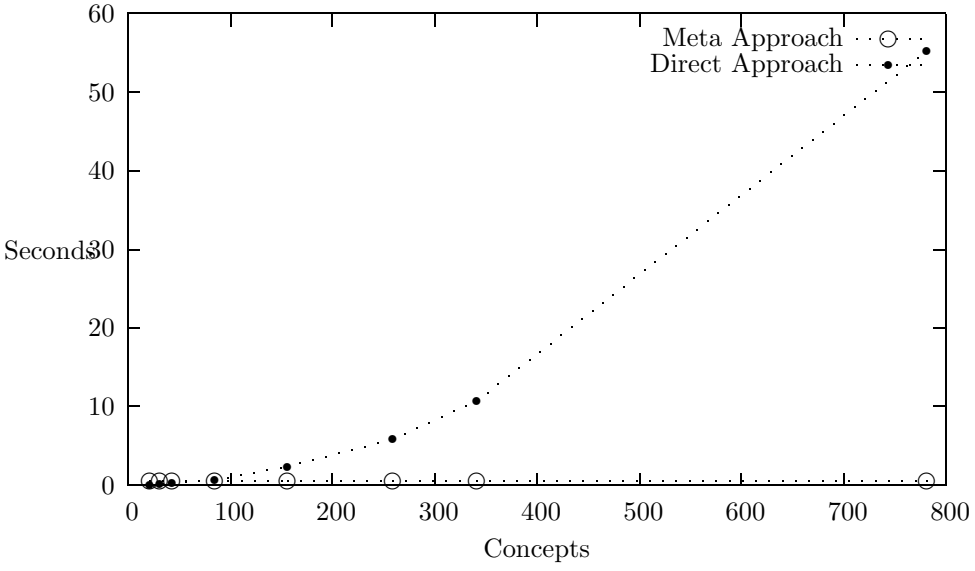

Fig. 2. Shows the time needed to preprocess the ontologies of Testcase 2 (see Section 5.3).

thus questionable whether efficient indexing mechanisms have been implemented. With a rising number of facts (tuples) such indexing mechanisms would be highly desirable. Indeed first tests indicate that CORAL lacks such mechanisms, which especially slows down the Meta Mapping approach as it has to perform a larger number of joins (because the average number of conjunctions per rule is higher) in comparison to the Direct Mapping approach. The Meta Mapping approach creates only few different relations with a comparatively large number of tuples. This is a procedure as meant for classical relational systems, which use indexing mechanisms like B-trees. In such systems we would thus expect only logarithmic increase in response time for the Meta Mapping approach. In fact we currently measure linear behavior for both approaches (as can be seen in Figure 3).

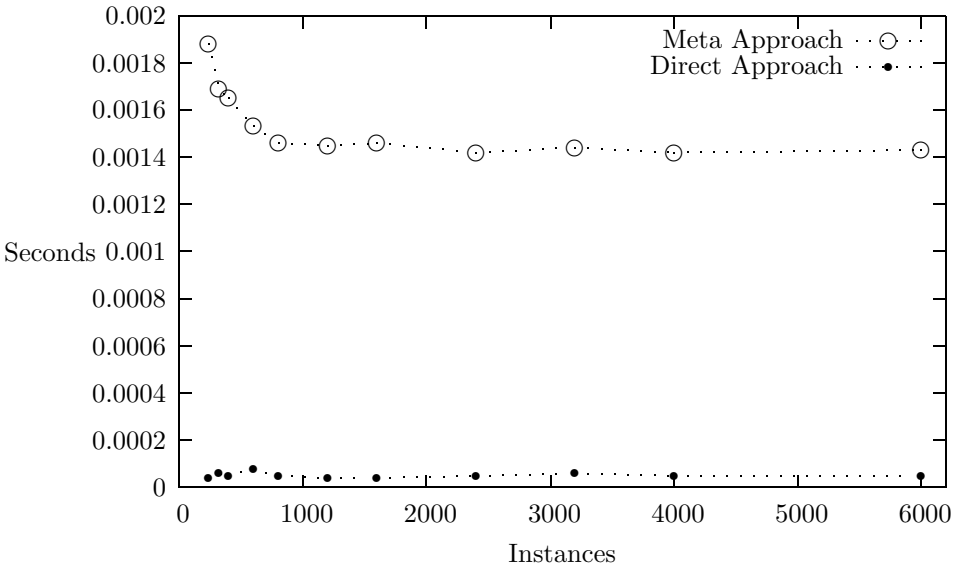

Fig. 3. Shows the linearity of the query time in Testcase 1 (See Section 5.3). The quotient of the number of instances and the query processing time is constant.

When having a look at current query time measurements we realize that the Direct Mapping approach answers faster (wherever it is able to deliver an answer at all) than the Meta Mapping approach. However we expect contrary results with a growing number of instances (which we cannot load into the database system using the Direct Mapping approach) and indexing performed on the relations.

In addition, CORAL performs the magic set transformation and other optimizations during the initial loading of the logic program. Other systems like the deductive database system LOLA perform these optimizations during query

processing. As you can see from the measurements performed this time is significantly higher (Figures 1 and 2) and growing exponatially (Figures 2) with the Direct Mapping approach. We can thus assume, that using the deductive database system LOLA (or any system working the same way) the query time would grow exponentially using the Direct Mapping approach but still remain linear using the Meta Mapping apporach.

6 Summary

The Semantic Web aims at extending the current Web in a way such that content of Web pages will not exclusively be meaningful for humans. Here, ontologies play a key role by providing machine processable semantics. A recent W3C working draft proposes OWL as ontology language for the Web. However, modeling in OWL requires quite some formal logical skills as well as elaborated reasoners which are not available of the shelf so far. In addition, reasoning systems very likely have to cope with much larger ontologies consisting of a huge number of individuals in the near future. However, most of todays knowledge processing systems are not designed to use secondary storage mechanisms and will therefore not meet upcoming scalability requirements. In a first step, it therefore seems promising to focus on the intersection of OWL with Logic Programs suitable to adopt logical database technology. Logical databases provide a declarative representation and querying language as well as efficient and persistent data storage. A mapping of OWL into Logic Programs has recently been proposed by [GHVD03]. In this Direct Mapping every class or property definition maps into a rule and every class or property instantiation into a fact of the resulting logic program. However, this approach has some representational as well as practical drawbacks. A conceptually disadvantage with far reaching consequences is the fact, that classes as well as properties are not accessible unless their names are explicitly known to the user

We therefore proposed a new approach using a mapping into a logical representation on an appropriate abstract meta level similar to a subset of the HiLog [CKW93] higher-order syntax. We showed that this Meta Mapping overcomes this limitations. Classes as well as properties are accessible as first class entities within this mapping allowing comfortable query formulation.

In our approach an ontology will be mapped into a logic program consisting of a constant number of rules with a linear growing number of facts proportional to the number of classes and properties. In the Direct Mapping the number of relations in the resulting logic program grows linear with the number of class and property definitions of the original ontology. When benchmarking both mappings it turned out that even a linear growth in relations of the Direct Mapping results in fatal performance during preprocessing while loading the program into the CORAL deductive database. Our experiments also observed a linear growth of time for class instance and subclass querying with an increasing number of individuals for both approaches. However, in case of the Meta Mapping approach

better results are expected for systems with secondary storage indexing mechanisms used in commercial systems today.

In consideration of the above we favor the Meta Mapping approach because of its significant conceptual advantages, higher expressivity and better performance for storing and evaluation of large scale real world ontologies in logical databases.

References

[CKW93] Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[FSS91] Burkhard Freitag, Heribert Schütz, and Günther Specht. *LOLA* - A Logic Language for Deductive Databases and its Implementation. In *Procc.* 2^{nd} *International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, pages 216 – 225, Tokyo, Japan, April 1991.

[GHVD03] Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programms: Combining Logic Programms with Description Logic. In *Proceedings of the* 12^{th} *International World Wide Web Conference*, Budapest, Hungary, May 2003.

[Gua98] Nicola Guarino. Formal Ontology and Information Systems. In *Proceedings of FOIS'98*, pages 3 – 15, Trento, Italy, June 1998.

[Hef03] Jeff Heflin. Web Ontology Language (OWL) Use Cases and Requirements. W3C Working Draft, March 2003.

[PSHH03] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. W3C Working Draft, March 2003.

[RSSS94] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL Deductive System. *VLDB Journal: Very Large Data Bases*, 3(2):161 – 210, 1994.

[Spe91] Günther Specht. LOLA, LDL, NAIL!, RDL, ADITI and STARDUST: A Comparision of Deductive Database Systems. Technical report, Institut für Informatik, TU München, 1991.

[vH02] Frank van Harmelen. The Complexity of the Web Ontology Language. *IEEE Intelligent Systems*, 17(2):71 – 72, March/April 2002.

[vHHH$^+$03] Frank von Harmelen, Jim Hender, Ian Horrocks, Deborah L. McGuiness, Peter F. Patel-Schneider, and Lynn A. Stein. OWL Web Ontology Language Reference. W3C Working Draft, March 2003.