Querying Objects with Description Logics

Martin Peim, Enrico Franconi, Norman W. Paton and Carole A. Goble
Department of Computer Science, University of Manchester
lastname@cs.man.ac.uk

Abstract

This paper presents an approach to answering queries over an ontology modelled using a description logic. The ontology acts as a global schema, providing a declarative description of the concepts of the domain, the instances of which are stored in (potentially many) object-wrapped sources. Queries are expressed using terms from the rich vocabulary of the ontology, and are translated into an equivalent calculus expression, which references only the objects available in the source databases. The query is then optimised on the basis of information from the ontology and the source databases.

1 Introduction

This paper presents an approach to answering queries over an ontology modelled using a description logic. Queries are formulated over the ontology in a query language which is itself a DL. The system translates these high-level user queries into queries over the definitions at the wrapper layer. In a wider technical context, the proposal presented in this paper is part of a collection of results on knowledge based query processing in distributed information systems. Such proposals can be classified as *global as view* (e.g., SIMS [1], OBSERVER [11]) or *local as view* (e.g. Information Manifold [10], DWQ [3], Picsel [8]), depending on how constructs from a specific source are related to those in the global model [13].

The *query evaluator* executes queries over a collection of object wrapped sources. Sources descriptions consist of a set of class declarations described using the ODMG's object definition language ODL—that is, an ODL *schema*—plus information needed for physical query optimisation, such as access paths (existence of indices, for example) and cost information. Extent names in the wrapper layer are associated with certain terms (which are referred to as *ground* terms) in the ontology by an *Ontology-to-Object-Model mapping*.

The translation process of a DL query can be broken down into several phases:

- The *query rewriting* phase rewrites the query as an expression containing only ground terms; in our case this is based on a global-as-view approach. Note, however, that a more sophisticated rewriting system based on the local-as-view technology could be used without affecting the rest of the system [13, 4]. Any ground term which appears in the rewritten query must be mapped to source constructs in the Ontology-to-Object-Model (OOM) mapping.

- The *query translation* phase translates the rewritten query into an expression in Fegaras' Monoid Comprehension Calculus [7], enriched with a *match*() operator to perform *object fusion*. The *match*() operator supports the reconstruction of a unique object by gathering sparse information coming from one or more sources. The translation process works in a compositional fashion, using the OOM mapping to translate DL terms, and handling logical connectives using a set of rules given in Section 4.

- The *semantic optimisation* phase takes the (possibly deeply nested) calculus expression generated by the query translation algorithm and performs simplifying transformations on it. The ontology is used at this stage to improve the calculus expression by identifying redundant generators (potential iterations) over source extents.

- The *calculus-to-algebra translation* and the *logical and physical optimisation* phases. From this point on, the translation system is an adaptation of Fegaras' OQL optimiser [7]. Firstly, a calculus expression is translated to a corresponding expression in a logical query algebra based on the nested-relational algebra. This logical algebra is then subject to heuristic optimisation by a logical optimiser, and then the logical algebra operators are replaced by operators in a physical algebra during physical optimisation. For lack of space, the logical and physical optimisers are not described further in this paper.

2 The Conceptual Model and Query Language

At the top level of the system, a unified and user-centred view of the source data is abstracted in a global conceptual schema expressed in the DL $\mathcal{ALCQI}$.: the ontology. Figure 1 contains a small example ontology in the bioinformatics field that will be used throughout the paper to illustrate query processing. Queries are formulated with respect to this i.e., the user may use any term from the ontology in the query, without knowing where the data actually is, how it is structured, and how it should be merged and reconciled to fit the global schema.

Given an ontology, such as that in Figure 1, a concept description can be taken as a query to retrieve all instances of the concept. For example, the following query asks for all proteins found in mammals:

$$\mathsf{protein} \sqcap \exists \mathsf{has\text{-}species}.\mathsf{mammal} \tag{15}$$

The following query, which is revisited later, asks for all proteins referred to in important journals.

$$\mathsf{protein} \sqcap \exists \mathsf{cited\text{-}in}.(\exists \mathsf{has\text{-}journal}.\mathsf{top\text{-}journal}) \tag{16}$$

The *query rewriting* phase rewrites the query as an expression containing only ground terms. The query rewriting task can be phrased in general terms as follows.

$$\mathsf{enzyme} \sqsubseteq \mathsf{protein} \sqcap \exists \mathsf{catalyses.reaction} \tag{1}$$

$$\mathsf{enzyme} \doteq \exists \mathsf{enz\text{-}protein}^{\top}.\top \tag{2}$$

$$\exists \mathsf{catalyses}.\top \sqsubseteq \mathsf{protein} \tag{3}$$

$$\mathsf{sp\text{-}protein} \sqsubseteq \mathsf{protein} \tag{4}$$

$$\mathsf{pir\text{-}protein} \sqsubseteq \mathsf{protein} \tag{5}$$

$$\mathsf{protein} \sqsubseteq (\exists \mathsf{has\text{-}sequence}.\top) \sqcap (\exists^{\leq 1} \mathsf{has\text{-}sequence}.\top) \tag{6}$$

$$\mathsf{sp\text{-}protein} \sqsubseteq (\exists \mathsf{sp\text{-}acc}.\top) \sqcap (\exists^{\leq 1} \mathsf{sp\text{-}acc}.\top) \sqcap (\forall \mathsf{has\text{-}species.species}) \tag{7}$$

$$\mathsf{pir\text{-}protein} \sqsubseteq (\exists \mathsf{pir\text{-}acc}.\top) \sqcap (\exists^{\leq 1} \mathsf{pir\text{-}acc}.\top) \sqcap (\forall \mathsf{cited\text{-}in.reference}) \tag{8}$$

$$\mathsf{species} \sqsubseteq (\exists \mathsf{common\text{-}name}.\top) \sqcap (\exists^{\leq 1} \mathsf{common\text{-}name}.\top) \sqcap (\exists \mathsf{latin\text{-}name}.\top) \sqcap (\exists^{\leq 1} \mathsf{latin\text{-}name}.\top) \tag{9}$$

$$\mathsf{mammal} \sqsubseteq \mathsf{species} \tag{10}$$

$$\mathsf{reference} \sqsubseteq (\exists \mathsf{has\text{-}author}.\top) \sqcap (\exists \mathsf{has\text{-}journal}.\top) \sqcap (\exists^{\leq 1} \mathsf{has\text{-}journal}.\top) \sqcap (\exists \mathsf{has\text{-}year}.\top) \sqcap (\exists^{\leq 1} \mathsf{has\text{-}year}.\top) \tag{11}$$

$$\mathsf{enz\text{-}entry} \sqsubseteq (\exists \mathsf{enz\text{-}protein}.\top) \sqcap (\forall \mathsf{enz\text{-}protein.sp\text{-}protein}) \sqcap (\exists \mathsf{enz\text{-}reaction}.\top) \sqcap (\exists \mathsf{has\text{-}journal}.\top) \sqcap (\exists^{\leq 1} \mathsf{has\text{-}journal}.\top) \sqcap (\exists \mathsf{has\text{-}year}.\top) \sqcap (\exists^{\leq 1} \mathsf{has\text{-}year}.\top) \tag{12}$$

$$\mathsf{journal} \doteq \exists \mathsf{has\text{-}journal}^{\top}.\top \tag{13}$$

$$\mathsf{top\text{-}journal} \sqsubseteq \mathsf{journal} \tag{14}$$

Figure 1: The example ontology.

Given a query Q, an ontology, and a set of view definitions that characterise the actual source data, reformulate the query into an expression, the rewriting, that refers only to the views, and provides the answer to Q. In our case, the view definitions correspond to the classes and relationships of the object wrapped sources. In this paper we do not describe the rewriting process, for we we do not provide any new result; in our practical application we use a pure global-as-view approach.

Once a query has been rewritten into an equivalent query containing only ground terms, it is important to check that the query is *safe* [6]. Essentially, a query (or concept) is considered safe if answering that query does not involve looking up information not referred to in the query. This is crucial to restrict the scope of a query. Our translation scheme in Section 4 only produces translations for safe queries. For example, the query $\forall R.C$ is unsafe because answering it involves, among other things, finding all individuals with no R fillers, and this information is not available from extents for R and C. Let Q be a concept expression which contains only ground concept names and has been rewritten into *negation normal form*. Then Q is safe if it has the form $\bot$, A (where A is a ground concept), $\exists R.C$ or $\exists^{\geq n} R.C$ (provided $n \geq 1$). It is unsafe if it has the form $\top$, $\neg A$, $\forall R.C$ or $\exists^{\leq n} R.C$. A conjunction is safe if and only if at least one of its conjuncts is safe. A disjunction is safe if and only if all of its disjuncts are safe. Note that, under this definition, a concept expression is safe if

and only if its negation is unsafe.

3 The Object Model and Calculus Queries

The query translator takes a DL query and translates it into a calculus expression over an object model. The object model is described in Section 3.1, and the calculus in Section 3.3.

3.1 The Object Model

The source databases are presented to the rest of the system by software wrappers in such a way that they can be seen as forming an object database, conforming to the ODMG data model [5]. From an implementation point of view, this is a practical choice because of its compatibility with CORBA and the fact the model is associated with well understood query processing techniques [7]. The structure of the objects returned by the wrappers is given by a schema in the ODMG's Object Definition Language (ODL).

Each ground concept in the ontology is viewed as a named persistent set of database objects. This set may be an extent over an ODL class or a set of values of some simple type like `String`. For the sake of brevity, we will refer to all such named collections as *source extents*. So that we can use the DL reasoner to assist in query optimisation (as described in Section 5), we require that each source name be represented by a name in the knowledge base, and we record any information about containment between sources in the ontology.

Figure 2 shows wrapper class definitions for the domain represented by the ontology in Figure 1. For example, the interface `Protein` represents protein data from the sources SwissProt and PIR, which are represented by the classes `SP_Protein` and `PIR_Protein`.

The attribute `SPAccessionNumber` in the source class `SP_Protein` corresponds to the accession number of a SwissProt entry. It is a unique identifier for the entry. The `species` attribute contains the set of species in which the protein can be found. Finally, `sequence` is a string representation of the protein's amino acid sequence.

Like SwissProt, PIR also identifies its protein entries with an accession number (see `PIR_Protein`). Note that PIR accession numbers are not the same as SwissProt accession numbers, so the attributes must be given different names and correspond to different roles at the DL level. The `references` attribute is a set of descriptions of references to the protein in the scientific literature. As with `SP_Entry`, the string `sequence` represents the amino acid sequence.

3.2 The Ontology-to-Object-Model Mapping

The OOM Mapping describes how ground DL concepts and roles relate to object model classes and relationships. For the example application, Table 1 gives the map-

```
interface Protein {
  attribute String sequence;
}
class SP_Protein (extent sp_proteins)
  extends Protein {
    attribute String SPAccessionNumber;
    attribute Set<Species> species;
    attribute String sequence;
}
class PIR_Protein (extent pir_proteins)
  extends Protein {
    attribute String PirAccessionNumber;
    attribute Set<Reference> references;
    attribute String sequence;
}
class Species (extent species) {
  attribute String common_name;
  attribute String latin_name;
}
class Reference (extent references) {
  attribute Set<String> authors;
  attribute String title;
  attribute String journal;
  attribute String year;
}
class EnzEntry (extent enz_entries) {
  attribute String enz_id;
  attribute Set<SP_Protein> enz_proteins;
  attribute Set<String> reactions;
  attribute Set<String> cofactors;
}
class Enz_catalyses_class (extent enz_catalyses) {
  attribute SP_Protein base;
  attribute String filler;
}
Set<Species> mammals
Set<String> top_journals
```

Figure 2: Declarations of source classes.

ping between DL concepts and source extents, and Table 2 shows the mapping of DL roles to class attributes.

DL roles are divided into *enumerated roles*, which are represented by the OOM mapping directly as sets of pairs, and *attribute roles*, which are represented as attributes of the base objects. An enumerated role R is represented by a set of source extents. Each of these is an extent e over a class of the form

```
class C (extent e){
   attribute T1 base;
   attribute T2 filler;
}
```

Concept	Source extents
protein	sp_proteins pir_proteins
sp-protein	sp_proteins
pir-protein	pir_proteins
enz-entry	enz_entries
species	species
reference	references
top-journal	top_journals
mammal	mammals

Table 1: Concept to source mapping for biological example

Role	Attribute	Cardinality
sp-acc	SPAccessionNumber	Single
pir-acc	PirAccessionNumber	Single
has-species	species	Multiple
has-sequence	sequence	Single
cited-in	references	Multiple
common-name	common_name	Single
latin-name	latin_name	Single
has-author	authors	Multiple
has-title	title	Single
has-journal	journal	Single
has-year	year	Single
enz-id	enz_id	Single
enz-protein	enz_proteins	Multiple
enz-reaction	reactions	Multiple
cofactor	cofactors	Multiple

Table 2: Attribute role mappings for biological example

The attribute names *base* and *filler* are used to refer to the first and second components of the binary relation represented by R. We say that e has type $T_1 \times T_2$.

The model contains a single enumerated role, catalyses, which represents the relationship between an enzyme and the reaction it catalyses. This role is mapped to the extent enz_catalyses of Figure 2, which is implemented as part of the wrapper on the Enzyme database.

An attribute role R is represented by an attribute name a_R. The attribute may be defined in several classes, and may have a different value type in each. For example, the fillers for a role like has-name may be simple strings in most classes, but structured objects (for example, botanical names of plants) in others. An attribute role can be either *single-valued* or *multiple-valued*. We require that a single-valued attribute role be represented by an attribute whose value type is a simple class name in each class which supports it, and that a multiple-valued attribute role be represented by an attribute whose value type is Set(T) for some class name T.

3.3 The Monoid Comprehension Calculus

The target language for the first stage of query translation is the Monoid Comprehension Calculus of Fegaras [7]. The calculus provides a uniform notation for collection types such as lists, bags and sets, based on the observation that the operations of set and bag union and list concatenation are *monoid* operations (that is, they are associative and have an identity element). Monoids for collection types are known as *collection monoids*. Operations like conjunctions and disjunctions on booleans and integer addition over collections can also be expressed in terms of so-called *primitive monoids*. A monoid comprehension has the form

$$\otimes\{e \mid q_1, \ldots, q_n\}. \tag{17}$$

The symbol $\otimes$ is a monoid operator, and determines the type of the comprehension. The expression e is called the *head* of the comprehension. Each q_i is a *qualifier*, which can either be a *generator* of the form $v \leftarrow e'$, where v is a variable and e' is a collection-valued expression, or a *filter* of the form p, where p is a predicate (a boolean-valued expression). Each variable v is assigned a type T, and the corresponding collection monoid must have an element type which is a subtype of T. We will usually omit the variable type from our notation, except where it needs to be emphasised. If q_i is a filter, then the head expression e and the q_j for $j > i$ can contain free occurrences of the variable v. The identity, or zero, element of the monoid whose operation is $\otimes$ is denoted by $\mathcal{Z}_\otimes$.

The primitive monoids used in examples below are: (i) The logical-and monoid $\wedge$. This is a simple monoid whose underlying type is boolean. The monoid operation is boolean conjunction and $\mathcal{Z}_\wedge = \textbf{true}$. (ii) The plus-monoid $+$. This is also a simple monoid whose underlying type is integer. The monoid operation is integer addition and $\mathcal{Z}_+ = 0$. The plus-monoid is used in examples in the form $+\{1 \mid \bar{s}\}$, to compute cardinalities of sets. The result of this expression is the number of distinct assignments to the generator variables in $\bar{s}$ which satisfy all the filters.

3.3.1 The Match-Union Monoid

A single individual belonging to the extension of a DL concept or query may be represented by several database instances (with distinct OIDs), coming from different sources. For example, protein instances may be represented in both the SwissProt and PIR sources. Thus, a query may have alternative answers if more than one choice of database instance is available for some of the relevant individuals. In order to support the reconstruction of a unique individual object from the sparse information coming from the same or different sources, we introduce a new boolean-valued operator *match*(x, y) which returns **true** if the database instances x and y represent the same individual.

This *match*$(\_, \_)$ operator is really a collection of operators *match*$_{S_1,S_2}(\_, \_)$, one for each pair of source databases S_1, S_2 (we can also divide it according to different object types returned by each source). We require that *match*$()$ defines an equivalence relation—that is, it must be reflexive, symmetric and transitive. We also assume that

distinct elements from the same source are intended to represent distinct individuals, so that if x and y come from the same source S, $\mathit{match}_{S,S}(x, y)$ reduces to $x = y$. The *match*() operator extends to tuples of objects in the obvious way. It can be interpreted as a simple equality test for domain values like integers and strings.

At the physical level, the implementation of *match*() for any given pair of sources may consist of a function which performs a comparison between certain key attributes of the objects concerned. Alternatively, if the two source classes do not have a common key, it might be necessary to use a binary table to associate the corresponding elements.

An answer to a query, then, is a set S of object references, such that

$$\text{for all } x, y \in S, \mathit{match}(x, y) = \textbf{false} \text{ unless } x = y. \tag{18}$$

Such sets are referred to as *match-sets*—they are still sets (rather than bags) even if we regard *match*() as equality.

In order to capture query answers as match-sets, comprehensions are written in terms of a collection monoid whose merge operation $\oplus$ (read as *match-union*) is like the set union operation but preserves the uniqueness condition (18). So, if S_1 and S_2 satisfy (18) then $S_1 \oplus S_2$ may be any set $W \subseteq S_1 \cup S_2$ of object references, such that for each $x \in S_1 \cup S_2$ there is precisely one $w \in W$ such that $\mathit{match}(x, w) = \textbf{true}$. For those elements of S_1 which match some element of S_2, we can choose which element to include in $S_1 \oplus S_2$. This choice can be made by the system on the basis of user preference or cost estimation or, if we have no preference, by taking the representative for S_1 whenever possible.

4 Translating Queries to Monoid Comprehensions

The rules given in this section show how to translate a safe $\mathcal{ALCQI}$ concept expression C into an expression E in the monoid comprehension calculus. To save space, we only consider expressions whose subexpressions are all safe. The implemented system [12] also deals with cases where this is not so. The modifications to the translation rules for these cases are indicated below. The rules constitute a compositional syntax-directed translation scheme. The expression E is a collection monoid comprehension using the monoid operation $\oplus$ described in Section 3.3.1. If the element type of this comprehension (the type of its head expression) is T we will say that E is a translation of C having type T.

The Empty Concept. The unsatisfiable concept $\bot$ is translated by the empty $\oplus$-monoid $\mathcal{Z}_\oplus$.

Atomic Terms. To translate a ground atomic concept (a concept name) A we consult the OOM mapping to find the set of database extent names which represent A. Each extent name has a type (a class name) T_i and refers to a set S_i of object references of type T_i. Let T be the most specific superclass of the T_i. Then $\bigoplus_i S_i$ is a translation of A having type T.

Concept	Translation	Type
$\exists R.C$	$\oplus\{r.\texttt{base} \mid r \leftarrow R', c \leftarrow C', \mathit{match}(c, r.\texttt{filler})\}$	T_1
$\exists^{\geq n} R.C$	$\oplus\{r.\texttt{base} \mid r \leftarrow R', +\{1 \mid s \leftarrow R', c \leftarrow C', \mathit{match}(s.\texttt{base}, r.\texttt{base}) \wedge \mathit{match}(s.\texttt{filler}, c)\} \geq n\}$	T_1

Table 3: Translation of enumerated roles.

For instance, in our example application the concept protein is mapped to the source extents `sp_proteins` and `pir_proteins`, and so it has the translation `sp_proteins` $\oplus$ `pir_proteins` (of type `Protein`).

Conjunctions. If C and D are safe concepts with translations C' and D' of type $T_{C'}$ and $T_{D'}$ then $C \sqcap D$ can be translated to either of the following, with types $T_{C'}$ and $T_{D'}$ respectively:

$$\oplus\{c \mid c \leftarrow C', d \leftarrow D', \mathit{match}(c, d)\} \tag{19}$$

$$\oplus\{d \mid c \leftarrow C', d \leftarrow D', \mathit{match}(c, d)\} \tag{20}$$

If we don't want to commit to choosing all our answers from one of C' and D' and we have a choice function $\mathit{choose}(x, y)$ which selects one of x and y according to some unspecified criteria, we can make a third translation with the type, T which is the most specific superclass of T_1 and T_2:

$$\oplus\{\mathit{choose}(c, d) \mid c : T \leftarrow C', d : T \leftarrow D', \mathit{match}(c, d)\} \tag{21}$$

Note the use of type specifiers for the variables c and d to emphasise that we are assigning subclass references to superclass reference variables. That is, we just take the $\oplus$ merge of C' and D' but interpret the references as having type T.

In the case where C is safe but D is unsafe, we must filter the instances of C for non-membership (up to $\mathit{match}()$) of the safe concept $\neg D$ rather than for membership of D.

Disjunctions. If C, D are safe concepts with translations C' and D' of type $T_{C'}$ and $T_{D'}$, let T be the most specific superclass of $T_{C'}$ and $T_{D'}$. Then

$$(\oplus\{c \mid c : T \leftarrow C'\}) \oplus (\oplus\{d \mid d : T \leftarrow D'\}) \tag{22}$$

is a translation of $C \sqcup D$ of type T.

Existentially Quantified and At-least Formulae. Existentially quantified formulae and at-least formulae are closely related, since $\exists R.C$ is equivalent to $\exists^{\geq 1} R.C$; they are handled together here.

- If R is an *enumerated role*, let $\{R'_i\}$ be the set of source tables for R and let the type of R'_i be $T_{i1} \times T_{i2}$. Let T_1 be the most specific superclass of the T_{i1}, let T_2 be the most specific superclass of the T_{i2}, and let

$$R' = \bigoplus_i (\oplus\{r \mid r : \langle \texttt{base} : T_1, \texttt{filler} : T_2 \rangle \leftarrow R'_i\}). \tag{23}$$

Concept	Translation	Type
$\exists R.C$	$\oplus\{d \mid d \leftarrow D_R, c \leftarrow C', \mathit{match}(d.a_R, c)\}$	T_1
$\exists R^-.C$	$\oplus\{d \mid d \leftarrow D_R, c \leftarrow C', \mathit{match}(d.a_R, c)\}$	T_2
$\exists^{\geq n} R^-.C$	$\oplus\{d.a_R \mid d \leftarrow D_R, +\{1 \mid c \leftarrow C', \mathit{match}(c, d)\} \geq n\}$	T_2

Table 4: Translation of single-valued roles.

Concept	Translation	Type
$\exists R.C$	$\oplus\{d \mid d \leftarrow D_R, f \leftarrow d.a_R, c \leftarrow C', \mathit{match}(f, c)\}$	T_1
$\exists R^-.C$	$\oplus\{f \mid d \leftarrow D_R, f \leftarrow d.a_R, c \leftarrow C', \mathit{match}(c, d)\}$	T_2
$\exists^{\geq n} R.C$	$\oplus\{d \mid d \leftarrow D_R, +\{1 \mid f \leftarrow d.a_R, c \leftarrow C', \mathit{match}(f, c)\} \geq n\}$	T_1
$\exists^{\geq n} R^-.C$	$\oplus\{f \mid d \leftarrow D_R, f \leftarrow d.a_R, +\{1 \mid e \leftarrow D_R, g \leftarrow e.a_R, c \leftarrow C', \mathit{match}(g, f) \wedge \mathit{match}(c, e)\} \geq n\}$	T_2

Table 5: Translation of multiple-valued roles.

Suppose C is safe and has a translation C'. Table 3 gives translations for enumerated roles.

Inverses of enumerated roles (for example, in $\exists R^-.C$) can be handled similarly, by exchanging the roles of `base` and `filler` (and of T_1 and T_2).

- If R is a *single-valued attribute role*, let $\{D_i\}$ be the set of domains for R. Let T_{i1} be the type of D_i and let T_{i2} be the value type of the attribute a_R in D_i. Let T_1 be the most specific superclass of the T_{i1} and let T_2 be the most specific superclass of the T_{i2}. Let

$$D_R = \bigoplus_i \oplus\{d \mid d : T_1 \leftarrow D_i\} \tag{24}$$

Then D_R represents the (potential) domain of the relation R.

Suppose C is safe and has a translation C'. Table 4 gives translations for single-valued roles. The translation of $\exists^{\geq n} R.C$ is equivalent to $\exists R.C$ if $n = 1$ and is empty if $n > 1$.

As an example of the translation of a single-valued attribute role, consider the translation of ∃has-journal.top-journal. According to the OOM mapping in Table 2, the role has-journal is a single-valued attribute role. It is mapped to the attribute `journal`, which is supported by the class extent `references`. The concept top-journal is mapped to the extent `top_journals`, as described in Table 1. Using the translation for $\exists R.C$ in Table 4, with renaming of variables, the translation is

$$\oplus\{r_1 \mid r_1 \leftarrow \texttt{references}, t \leftarrow \texttt{top\_journals}, \mathit{match}(r_1.\texttt{journal}, t)\} \tag{25}$$

- If R is a *multiple-valued attribute role*, let $\{D_i\}$, T_{i1}, T_{i2} T_1 and T_2 be defined as for single-valued attributes above, except that the value type of a_R in T_{i1} is now $\mathsf{Set}(T_{i2})$

Suppose C is safe and has a translation C'. Table 5 gives translations for multiple-valued roles.

An expression such as $\exists R.C$ or $\exists^{\leq n} R.C$ where C is unsafe is translated in a similar manner to the above, except that, instead of checking that objects related to a given instance of the domain of R are *match*()-equivalent to instances of C we must check that they are not equivalent to any instace of the safe concept $\neg C$.

4.1 Example Translation

As an example of the query translation process, consider query (16), which asks for all proteins referred to in important journals. To translate this query, we first translate the subquery:

$$\exists \mathsf{has\text{-}journal}.\mathsf{top\text{-}journal}. \tag{26}$$

This gives rise to the calculus expression (25) described above. Proceeding in this way we obtain the monoid comprehension:

$$\begin{array}{l} \oplus\{p_2 \mid p_2 \leftarrow \texttt{sp\_proteins} \oplus \texttt{pir\_proteins}, \\ \quad p_3 \leftarrow \oplus\{p_1 \mid p_1 \leftarrow \texttt{pir\_proteins}, \\ \qquad\qquad r_2 \leftarrow p_1.\texttt{references}, \\ \qquad\qquad r_3 \leftarrow \oplus\{r_1 \mid r_1 \leftarrow \texttt{references}, \\ \qquad\qquad\qquad\quad t \leftarrow \texttt{top\_journals}, \\ \qquad\qquad\qquad\quad \mathit{match}(r_1.\texttt{journal}, t)\}, \\ \qquad\qquad \mathit{match}(r_2, r_3)\}, \\ \quad \mathit{match}(p_2, p_3)\}, \end{array} \tag{27}$$

supposing that the system decides to use both sources for the concept protein. This rather unwieldy form can be considerably simplified by the methods of Section 5.

5 Optimisation

For the most part, the optimisation of the queries that result from the translation process described in Section 4 follows that of Fegaras' optimiser [7]. Following translation, a normalisation algorithm rewrites the comprehension into a normal form. The normalisation rules are given in Section 5.1. In an extension to Fegaras' normalisation algorithm, certain optimisations are made during the normalisation process, to remove unnecessary iterations or generators. Unlike [9]—where both the constraints introduced by the ODL schema and the OQL queries are translated into a Datalog program, and semantic optimisation is performed by adding extra conditions or "residues" coming from the integrity constraints to the query optimiser—and unlike [2]—where the system computes the complete explicit semantic expansion of the original query—we base the semantic optimisations on *oracle* calls to the DL reasoner from the standard Fegaras' OQL optimiser. These calls check containment

$$\langle A_1 = e_1, \ldots, A_n = e_n \rangle.A_i \longrightarrow e_i \tag{28}$$

$$\otimes\{e \mid \bar{q}, v \leftarrow \mathcal{Z}_{\oslash}, \bar{s}\} \longrightarrow \mathcal{Z}_{\otimes} \tag{29}$$

$$\odot\{e \mid \bar{q}, v \leftarrow e_1 \oplus e_2, \bar{s}\} \longrightarrow (\odot\{e \mid \bar{q}, v \leftarrow e_1, \bar{s}\}) \odot (\odot\{e \mid \bar{q}, v \leftarrow e_2, \bar{s}\}) \tag{30}$$

$$+\{e \mid \bar{q}, v \leftarrow e_1 \oplus e_2, \bar{s}\} \longrightarrow (+\{e \mid \bar{q}, v \leftarrow e_1, \bar{s}\}) + (+\{e \mid \bar{q}, v \leftarrow e_2, \wedge\{\neg \mathit{match}(v, w) \mid w \leftarrow e_1\}, \bar{s}\}) \tag{31}$$

$$\otimes\{e \mid \bar{q}, \vee\{\mathit{pred} \mid \bar{r}\}, \bar{s}\} \longrightarrow \otimes\{e \mid \bar{q}, \bar{r}, \mathit{pred}, \bar{s}\} \tag{32}$$

$$\otimes\{e \mid \bar{q}, v \leftarrow \oplus\{e' \mid \bar{r}\}, \bar{s}\} \longrightarrow \otimes\{e[e'/v] \mid \bar{q}, \bar{r}, \bar{s}[e'/v]\} \tag{33}$$

$$*\{*\{e \mid \bar{r}\} \mid \bar{s}\} \longrightarrow *\{e \mid \bar{s}, \bar{r}\}, \tag{34}$$

Figure 3: Normalisation rules for the calculus.

$$\oplus\{e \mid \bar{q}, v \leftarrow X, \bar{r}, w \leftarrow Y, \bar{s}, \mathit{match}(v, w) \wedge p\} \longrightarrow \oplus\{e[v/w] \mid \bar{q}, v \leftarrow X, \bar{r}, \bar{s}[v/w], p[v/w]\}, \quad \text{if } X \sqsubseteq Y \tag{36}$$

$$\oplus\{e \mid \bar{q}, v \leftarrow X, \bar{r}, w \leftarrow Y, \bar{s}, \mathit{match}(v, w) \wedge p\} \longrightarrow \oplus\{e[v/w] \mid \bar{q}, v \leftarrow Y, \bar{r}, \bar{s}[v/w], p[v/w]\}, \quad \text{if } Y \sqsubseteq X \tag{37}$$

$$\oplus\{e \mid \bar{q}, v \leftarrow X, \bar{r}, w \leftarrow Y, \bar{s}, \mathit{match}(v, w) \wedge p\} \longrightarrow \emptyset, \quad \text{if } X \sqcap Y \doteq \bot. \tag{38}$$

$$\oplus\{e \mid \bar{q}, x \leftarrow X \oplus Z, \bar{r}, p\} \longrightarrow \oplus\{e \mid \bar{q}, x \leftarrow X, \bar{r}, p\}, \quad \text{if } \mathit{can\text{-}restrict}(x, X, p, \mathit{concat}(q, r)) \tag{39}$$

Figure 4: Semantic optimisation rules for the calculus.

between sub-queries given the semantic information specified in the ontology. The optimisation rules are given in Section 5.2.

5.1 Normalisation

The first stage of Fegaras' optimiser [7] is a normalisation process which does some unnesting of nested comprehensions. The process results in a canonical form which is (in our case) a $\oplus$-merge of comprehensions of the form

$$\oplus\{e \mid v_1 \leftarrow \mathit{path}_1, \ldots, v_n \leftarrow \mathit{path}_n, \mathit{pred}\}, \tag{35}$$

where each path_i is a database extent name or an expression of the form $v.a_1 \ldots a_n$, where v is a bound variable and the a_i are attribute names. Note that the head e and the predicate $pred$ may still contain nested comprehensions, though these will also be in canonical form. The normalisation rules needed to convert the comprehension expressions produced by the algorithm in Section 4 are given in Figure 3. In the figure, $\otimes$ and $\oslash$ may be any of the monoid operations $\oplus$, $\vee$, $\wedge$ or $+$, $*$ may be $\vee$, $\wedge$ or $+$, and $\odot$ may be $\oplus$, $\wedge$ or $\vee$. The notation $e[e'/v]$ denotes the result of substituting e' for the free occurrences of v in e. Further details on the normalisation of the monoid calculus can be obtained from [7].

5.2 Semantic Optimisation

The translations given in Section 4 are applicable to any concept expression. However, in certain circumstances more efficient translations can be produced by exploiting knowledge about the types returned by translations of subexpressions and the containment relationships stored within the ontology.

In order to be meaningful for query optimization purposes, the knowledge stored in the ontology should state constraints on the extensional state of the database. This means that the database must strictly conform to the constraints, and it can not contain incomplete information.

For example, the concept $C \sqcap \exists R.D$ (where R is a multiple-valued attribute role, C is translated as C' and D as D') is translated to a monoid comprehension which, after normalisation (ignoring for the moment the fact that C' and D_R may be $\oplus$-unions), looks like

$$\oplus\{c \mid c \leftarrow C', x \leftarrow D_R, f \leftarrow x.a_R, d \leftarrow D', \mathit{match}(c, x), \mathit{match}(f, d)\}. \tag{40}$$

However, if the type of C' is such that the elements can be guaranteed to be in D_R, the iteration over x can be dispensed with. In this case, each instantiation of the variable c has its own set $c.a_R$ of R-fillers and the query can be answered by the comprehension

$$\oplus\{c \mid c \leftarrow C', f \leftarrow c.a_R, d \leftarrow D', \mathit{match}(f, d)\} \tag{41}$$

The optimisation rules (36) and (37) in Figure 4 achieve the required simplification.

Similarly, if a comprehension has two generators whose variables are supposed to match but whose domains are known to be incompatible from the ontology, then the comprehension is empty. This is captured by rule (38) in Figure 4.

Another optimisation can be applied to a comprehension C that contains a generator of the form $x \leftarrow X \oplus Z$. If the predicate of C implies matches between x and other variables, then it may be that the intersection of the ranges of those variables is contained in X. In that case we can restrict x to range only over X. The formalisation of this rule (rule (39) in Figure 4) refers to the predicate *can-restrict* defined as follows. Let x be a variable, X a union of extents, p a predicate and $\bar{r}$ a sequence of generators. Let $\{y_i\}$ be the set of expressions which are related to x by *match*() conjuncts in p, not including x itself. The y_i are the elements (excluding x) of the connected component of the graph defined by the *match*() conjuncts in p. Each y_i is either a variable or a path expression of the form $z.a_1 \ldots a_n$ where z is a variable and the a_j are attributes. So each y_i has a type which corresponds to a DL concept Y_i. Let X' be the DL concept corresponding to X. Then *can-restrict*$(x, X, p, \bar{q})$ is true if $(\prod_i Y_i) \sqsubseteq X'$ (which we can find out from the DL classifier) and false otherwise.

5.3 Simplification Example

As an example of query simplification, we can consider the translation (27) of query (16) from Section 4. This form immediately admits a simplification by rule (39), since the type of p_3 is `PIR_Protein` which corresponds to the concept pir_protein so that the

sp_proteins summand in the domain of p_2 can be eliminated. Normalisation then yields

$$\begin{aligned}\oplus\{p_2 \mid\ & p_2 \leftarrow \texttt{pir\_proteins}, p_1 \leftarrow \texttt{pir\_proteins},\\ & r_2 \leftarrow p_1.\texttt{references}, r_1 \leftarrow \texttt{references},\\ & t \leftarrow \texttt{top\_journals}, \mathit{match}(r_1.\texttt{journal}, t) \wedge\\ & \mathit{match}(r_2, r_1) \wedge \mathit{match}(p_2, p_1)\}.\end{aligned} \tag{42}$$

Two applications of rule (36) then eliminate the variables p_1 and r_1, leaving the form

$$\begin{aligned}\oplus\{p_2 \mid\ & p_2 \leftarrow \texttt{pir\_proteins}, r_2 \leftarrow p_2.\texttt{references},\\ & t \leftarrow \texttt{top\_journals}, \mathit{match}(r_2.\texttt{journal}, t)\}.\end{aligned} \tag{43}$$

Note that the *match*() comparison which remains is between values of type String and so it will be evaluated by a simple equality test.

6 Conclusions

The provision of knowledge-based information integration systems has been a focus of research activity for a considerable period, as it holds out the hope that high-level, declarative representations of resources can be used for schema reconciliation and query answering. This paper seeks to contribute to this line of research by bringing together recent results on expressive description logics and object database query processing to provide more expressive modelling and query processing to global-as-view query systems. The paper not only shows how queries over an $\mathcal{ALCQI}$ ontology can be mapped to the monoid calculus for evaluation, but has also demonstrated how information from the ontology can be used to simplify the resulting calculus expression.

References

[1] Y. Arens, C.A. Knoblock, and W-M. Shen. Query Reformulation for Dynamic Information Integration. *J. Intelligent Information Systems*, 6(2/3):99–130, 1996.

[2] S. Bergamaschi, D. Beneventano, C. Sartori, and M. Vincini. Odb-qoptimizer: A tool for semantic query optimization in oodb. In *Proc. of the Thirteenth International Conference on Data Engineering (ICDE'97)*, page 578, 1997.

[3] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Information integration: Conceptual modeling and reasoning support. In *Proc. of the 6th Int. Conf. on Cooperative Information Systems (CoopIS'98)*, pages 280–291, 1998.

[4] D. Calvanese, G. De Giacomo, M. Lenzerini, and Moshe Y. Vardi. View-based query processing and constraint satisfaction. In *Proc. of the 15th IEEE Sym. on Logic in Computer Science (LICS 2000)*, 2000.

[5] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[6] P. T. Devanbu. Translating description logics to information server queries. In *CProc. of the Second International Conference on Information and Knowledge Management (CIKM'93)*, pages 256–263, 1993.

[7] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 2001. (to appear).

[8] F. Goasdoue, V. Lattes, and M-C. Rousset. The use of CARIN language and algorithms for information integration: the picsel system. *International Journal on Cooperative Information Systems*, 2000.

[9] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proc. of the Thirteenth International Conference on Data Engineering (ICDE'97)*, pages 444–453, 1997.

[10] A.Y. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information System s. *J. Intelligent Information Systems*, 5:121–143, 1995.

[11] E. Mena, A. Illarramendi, V. Kashyap, and A.P Sheth. OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. *Distributed and Parallel Databases*, 8(2):223–271, 2000.

[12] M. Peim, E. Franconi, N. W. Paton, and C. A. Goble. Query processing with description logic ontologies over object-wrapped databases. Technical report, Dept. of Computer Science, University of Manchester, UK, 2001.

[13] J. D. Ullman. Information integration using logical views. In *Proc. of the 6th International Conference on Database theory (ICDT'97)*, volume 1186, pages 19–40. Springer-Verlag, 1997.