Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-11-01

An Efficient Direct Implementation of Equivalence Relations in OWL

Hans-Ulrich Krieger

October 2011

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80	Stuhlsatzenhausweg 3	Robert-Hooke-Str. 5
D-67608 Kaiserslautern	D-66123 Saarbrücken	D-28359 Bremen, Germany
Tel.: + 49 (631) 205 75-0	Tel.: + 49 (681) 302-5151	Tel.: +49 (421) 218-64 100
Fax: + 49 (631) 205 75-503	Fax: + 49 (681) 302-5341	Fax: +49 (421) 218-64 150

E-Mail: reports@dfki.de WWW: http://www.dfki.de

An Efficient Direct Implementation of Equivalence Relations in OWL

Hans-Ulrich Krieger

DFKI-RR-11-01

This work and the implementation of *HFC* started in 2007 when I worked for the EU-funded project MUSING (2006–2010; FP6 IST 27097), but has also benefitted from work conducted in the CogX project (cogx.eu; FP7 ICT 215181). Writing up this report is supported by the MONNET project (monnet-project.eu; FP7 ICT 248458).

ISSN 0946-008X

An Efficient Direct Implementation of Equivalence Relations in OWL

Hans-Ulrich Krieger
Language Technology Lab
German Research Center for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
`krieger@dfki.de`

Abstract

This paper presents a direct implementation of the three equivalence relations for TBox, RBox, and ABox in OWL and has been realized in the forward chaining engine *HFC*. The technique obviates the combinatorial explosion in a semantic repository during materialization, when applying the OWL entailment rules from ter Horst (2005) that are used in, e.g., Jena or OWLIM. Although the approach requires some work when starting up a repository and and querying its content, it massively pays off in the end by showing a smaller memory footprint and allowing faster inferences, as our measurements have shown. As a nice side effect of this approach, the "cleanup" of ontologies can even lead to smaller repositories, as more equivalence relation instances are added. Our decision to add such functionality in *HFC* was motivated by experiences we have gained in former projects that have dealt with information extraction from natural language texts, represented in description logic ontologies.

1 Introduction

This paper presents a direct implementation of the three equivalence relations used for the TBox, RBox, and ABox in the language specification of OWL, viz., `equivalentClass`, `equivalentProperty`, and `sameAs`. The approach described here has been realized in the forward chaining engine *HFC* that we have developed over the last years and that is comparable to popular engines, such as OWLIM [4] or Jena [12].
The proposed technique obviates the combinatorial explosion attributed to equivalence relations in a semantic repository during materialization, when applying the OWL entailment rules from [14]. Although the approach requires some work when (i) starting up a repository (cleaning up data, rewriting rules) and (ii) querying its content (replacing individuals by their proxies, and vice versa), it pays off in the end by showing a smaller memory footprint and allowing faster inferences than the standard brute-force approach which multiplies out everything.
Our decision to add such functionality in *HFC* was motivated by common experiences we have gained in different projects, viz., querying structured knowledge sources obtained by information extraction from natural language texts [2, 7], anchoring facts/RDF triples in time [6], and performing description logic rule-based reasoning over uncertain sensor data [9].

The structure of this paper is as follows. In the next section, we motivate why it is preferable to have an optimized implementation of equivalence relations in OWL. We then describe the implementation of our approach: the disjoint-set data structure, the cleanup phase, the rule rewriting mechanism, and the two querying modes. This section also contains a short description of *HFC*, the forward chaining engine on which we base our implementation. After that, measurements in the space and time domain are performed against an ontology that is equipped with a varying number of equivalence relation instances. These data sets are then materialized using *HFC*, both with and without the optimized equivalence relation implementation. We finish this paper by giving a summary, presenting further remarks, and relate our system to others.

2 Motivation: Equivalence Relations in OWL

In line with expressive description logics, OWL provides so-called *property characteristics*, such as `TransitiveProperty` or `SymmetricProperty`. However, there is no way to directly characterize a property as being an *equivalence relation* which is a combination (a conjunction) of three characteristics:

> A relation R on a set S is an *equivalence relation* if R is *reflexive*, *symmetric*, and *transitive* [1], i.e.,
>
> - $\forall s \in S \,.\, s\,R\,s$
> - $\forall s, t \in S \,.\, s\,R\,t \rightarrow t\,R\,s$
> - $\forall s, t, u \in S \,.\, s\,R\,t \wedge t\,R\,u \rightarrow s\,R\,u$

We define the *equivalence class* $[s]$ of $s \in S$ to be the set

$$[s] := \{t \in S \mid s\, R\, t\}$$

OWL comes up with three *predefined* equivalence relations that express axioms over individuals from the TBox, RBox, and ABox of an ontology, viz.,

1. `equivalentClass`
2. `equivalentProperty`
3. `sameAs`

(1.) and (2.) are often useful when it comes to merging different ontologies, e.g., when interfacing an upper ontology with a domain-specific ontology, or by bringing two different upper ontologies together through bridging concepts and properties. In [6], we have shown how an upper ontology (viz., PROTON) can be equipped with a concept of time. This was achieved by adding the following axiom to the union of the two ontologies (PROTON and perdurant ontology):

$\mathsf{psys{:}Entity} \equiv \mathsf{fourd{:}TimeSlice}$

$\mathsf{TimeSlice}$ here "adds" time to the very general PROTON class Entity that possesses several hundred subclasses. Such axioms are a good thing when information from different sites and people comes together, but clearly slows down the performance of a semantic repository, both in the time and space domain of reasoning and querying. Given the above example plus the information

$\mathsf{pupp{:}Company} \sqsubseteq \mathsf{psys{:}Entity}$
$\mathsf{apple} : \mathsf{pupp{:}Company}$

we can legally infer that

$\mathsf{pupp{:}Company} \sqsubseteq \mathsf{fourd{:}TimeSlice}$
$\mathsf{apple} : \mathsf{fourd{:}TimeSlice}$

but also

$\mathsf{psys{:}Entity} \equiv \mathsf{fourd{:}TimeSlice}$
$\mathsf{fourd{:}TimeSlice} \equiv \mathsf{fourd{:}TimeSlice}$
$\mathsf{psys{:}Entity} \equiv \mathsf{psys{:}Entity}$

is the case, due to the symmetry and reflexivity of $\equiv$. Especially the transitive nature of equivalence relations makes forward expansion dangerous. The example here, however, does not involve more than two classes, lying in the same equivalence class, and in fact, when merging TBoxes and RBoxes from different ontologies, more than two classes or properties can be rarely found here. This, however, is usually **not** the case when we turn our attention to the ABox which interacts via entailment rules with axioms from TBox and RBox.

`sameAs` (plus `differentFrom`) is great to add some "safety" to an open world by identifying individuals (or making them definitely different). `sameAs` thus is often used in ontology-based information extraction systems which usually introduce a new URI for each new mention of a potential individual in a natural language text (e.g., pronoun, proper noun, noun phrases, even sentential phrases). In a reference resolution phase, several of those individuals are recognized to be identical, say, for example

$\{\mathsf{apple}\} \equiv \{\mathsf{apple\text{-}1}\}$
$\{\mathsf{apple\text{-}1}\} \equiv \{\mathsf{apple\_computer\_inc\text{-}1}\}$
$\{\mathsf{apple\_computer\_inc\text{-}1}\} \equiv \{\mathsf{apple\_inc\text{-}1}\}$
$\{\mathsf{apple\_inc\text{-}1}\} \equiv \{\mathsf{steve\_jobs\_company\text{-}1}\}$

These four axiom result in an equivalence class of five URIs. Overall, materialization leads to 25 `sameAs` statements. In general, given an equivalence class $[u]$ for a URI u, we obtain $|[u]|^2$ statements.
The bad thing now is that the identification of ABox individuals interact with "ordinary" ABox statements, perhaps coming from the initial ontology, say

apple : Company
(apple, steve_jobs) : hasCeo
(apple, steve_jobs) : foundedBy
(apple, steve_wozniak) : foundedBy
(apple, ronald_wayne) : foundedBy

These five statements, plus the the five elements from [apple] result in 20 *new* statements. In the worst case, a binary ABox relation instance

$$(s, o) : p$$

can lead to a lot of *new* statements, viz.,

$$|[s]| \times |[o]| \times |[p]| - 1$$

Given all this, it seems desirable to avoid such an explosion of facts.
In the remainder of this paper, we present a direct implementation of equivalence relations in *HFC*, together with measurements showing that our proposal is far superior to the native implementation.

3 Implementing Equivalence Relations in *HFC*

As already mentioned above, OWL provides three different equivalence relations for the TBox, RBox, and ABox of an ontology, viz., `equivalentClass`, `equivalentProperty`, and `sameAs`.[1] These relations are used in the entailment rules [14] which are applied in forward engines during the materialization phase of a repository (e.g., in OWLIM, Jena, or *HFC*).
Not only are these relations used in entailment or even user-defined rules, they can also appear when a repository starts up with initial data, when several ontologies are merged, or when ABox relation instances are added at runtime by an external process.
Finally, when querying a repository, equivalence relation instances might appear as (potentially underspecified) clauses in the `WHERE` part of a SPARQL query (or any other query language).

[1] It is worth noting that our treatment can not only be applied to the three OWL relations, but might also be used to address general equivalence relations in an OWL setting, e.g., by explicitly characterizing a property to be an `equivalenceRelation`, or by making a property an equivalence relation when it is reflexive (now in OWL 2), symmetric, and transitive at the same time.

Before focusing on these three problem areas, we start with a short description of *HFC* which we have used as the platform for our implementation.

3.1 *HFC*

Usually, bottom-up forward chaining is employed to carry out (all possible) inferences at compile time, so that querying information reduces to an indexing problem at runtime. The process of making implicit information explicit is often called *materialization* or computing the *deductive closure* of a set of ground atoms $\mathcal{A}$ w.r.t. a set R of universally-quantified implications or *if-then* rules

$$B \rightarrow H$$

Bottom-up here means that one starts from the ground atoms to which the rules are applied, contrary to top-down approaches which start with a goal (the head H) and potentially hypothesize intermediate goals that can hopefully be satisfied by ground atoms finally (Prolog's strategy). The body and the head of a rule consist of a set of clauses, interpreted *conjunctively*. In *HFC*, clause arguments are either constants (URIs or XSD atoms) or variables.
Closure computation can be characterized as the computation of the *least fixpoint* of a certain *monotonic* function over the complete lattice $\wp(\mathcal{A})$ of the set of all ground atoms $\mathcal{A}$. Forward chaining, as we used it here, can be seen as *model building* over the Herbrand interpretation of a function-free definite program (Horn logic as used in Prolog). In general, model builders are systems that try to construct a finite model for a given theory (usually, a set of first-order formulae).
In order to make forward chaining scalable, *HFC* applies several optimization techniques that are realized as a sequence of filter stages in order to avoid useless RHS instantiations. This comes as a side product of the fact that closure computation is a monotonic operation: new ground atoms are only added, nothing is deleted. Consider, for instance, a rule

$$r = (b_1\, b_2 \rightarrow H)$$

and assume that r is currently applied in iteration n of the closure computation. Due to the monotonicity argument, matching candidates M^n from $\mathcal{A}$ for the LHS variables of rule r in iteration n can be decomposed into those which are brand new at n and those which come from iteration $n-1$:

$$M^n = N \uplus M^{n-1}$$

Since bindings for the variables of individual clauses are actually tables, computing a binding for all LHS variables effectively reduces to a *natural join* $\bowtie$, known from data base theory. Given the distinction *new vs. old* already mentioned, we can compute all possible bindings for $b_1\, b_2$ from the individual bindings, given N and M^{n-1}:

$$M^n(b_1\, b_2) \;=\; N(b_1) \bowtie N(b_2) \,\cup\, N(b_1) \bowtie M^{n-1}(b_2) \,\cup\, M^{n-1}(b_1) \bowtie N(b_2)$$

This optimization[2] massively speeds up forward chaining, since useless bindings, leading to already instantiated triples, are no longer generated. In our case here,

$$M^{n-1}(b_1) \bowtie M^{n-1}(b_2)$$

is **not** computed anymore, and the set of those bindings are by far the largest in size, when closure generation n increases. Intermediate results are even *memoized* in case more than two tables are involved in order to avoid recomputation of already computed results. This techniques not only applies to individual clauses, but also to larger parts, so-called (LHS) clusters. *HFC* has included further optimizations, e.g.,

- bindings are shared over "similar" clause, even between different rules;
- OWL equivalence relation instances in rules are efficiently handled through offline rule rewriting and a union-find data structure (**this paper**);
- the LHSs of rules are reordered to faster compute matching candidates;
- the processing of individual rules can be parallelized at each fixpoint iteration step by specifying the number of processor cores;
- efficient data structures, such as open-address hash tables, integer arrays for triples, specialized sets with strategy objects to support binding/table projection, etc., are used.

HFC has implemented several extensions that are not available in comparable systems, such as OWLIM, some of them have eased the direct implementation of equivalence relations in OWL:

- replacement of triples by more general tuples,
- **possibility to add arbitrary tests to the LHS of a rule**,
- **possibility to add arbitrary actions to the RHS of a rule**,
- incorporation of aggregation rules,
- incorporation of metric linear, potentially underspecified calendar time.

HFC efficiently handles ABoxes with millions of facts and provides means to work with thousands of medium-sized ABox in parallel, an important feature that we employ in the forward branching time approach, described in [9]. The memory measurements that we present in the next section are related to *HFC*, running in reasoning mode. To speed up rule execution, large sets of triples are maintained for each rule that keep the distinction between *old vs. new*, as explained above. When used as a pure storage engine, the memory footprint of *HFC* is much smaller, e.g., less than 26GB for storing and accessing 100,000,000 triples.

[2]We have already applied this idea more than 10 years ago in a quite different area, viz., context-free approximation of unification-based grammars; see, e.g., [3].

3.2 Disjoint-Set Data Structure

A *disjoint-set data structure* implements a representation for a collection of disjoint dynamic sets [1]. Each set represents an equivalence class $[s]$ and s is called the *representative* (or *proxy*) of $[s]$. In our case, these sets are sets of positive natural numbers (four-byte integers I, or even eight-byte long integers), representing URIs. We maintain two mappings in our implementation, viz.,

1. $\mathit{uriToProxy} : I \mapsto I$ which maps an URI to its representative, and
2. $\mathit{proxyToUris} : I \mapsto 2^I$ which enumerates those URIs lying in the same equivalence class, given the representative (the proxy).

Both mappings are updated dynamically within cleanup phases when equivalence relation instances (ERIs) are processed. The first mapping is employed during input and output when new facts are imported or when a repository is queried and URIs need to be replaced by their proxies. The second mapping is solely used at query time in one of the output modes (proxy vs. multiply-out mode). Even though a repository does not contain any ERIs *after* a cleanup (with the notable exception of reflexive proxy instances, see section 3.3), the first mapping is still accessed and extended during the materialization phase when the rules are applied. Why this is so will become apparent in section 3.4.

3.3 Cleaning Up a Repository

When setting up a repository, an initial ontology might contain ERIs. The ontology, which we have used for our measurements below, comes up with such statements, e.g.,

ltw:Speech_Corpora $\equiv$ ltw:Spoken_Language_Corpora

At runtime, e.g., during information extraction, `sameAs` statements are likely to be introduced (see apple-steve_jobs example above). In order to avoid the combinatorial explosion during materialization, we need to *clean up* the whole data. There are basically two strategies to achieve this,

1. either each time an ERI is detected when new information is added,
2. or at a well-defined stage when all information has been imported.

We have opted for both ways, however, in different situations. When reading in facts (RDF triples in *HFC*, or even general tuples) from a file, the equivalence class reduction is applied only at the very end of the import. Similarly, materialization is followed up by a cleaning phase. When only single facts are added by an external process, the user however decides (through the use of two specific methods) whether data cleaning should be immediately applied or not.

Up to this point, we have used description logic syntax. From now on, we move over to the RDF-based triple notation [10]

subject predicate object

in order to be "closer" to the implementation, but also to avoid the description logic distinction between unary and binary predicates in the algorithm below. During the cleanup phase, several things happen (the implementation is more intelligent than the pseudo code suggests, e.g., we do not simply iterate over T):

```
Cleanup( ) ≡
01 global $T$
02 global uriToProxy
03 local $P := \{\texttt{equivalentClass}, \texttt{equivalentProperty}, \texttt{sameAs}\}$
04 for each $t = (s\ p\ o) \in T$
05    if $p \in P$
06       $T := T \setminus \{t\}$
07       UpdateUriToProxy(s, o)
08       UpdateProxyToUris(s, o)
09       UpdateUriToEqRel(s, p)
10 for each $(k, v) \in$ uriToProxy
11    for each $t = (s\ p\ o) \in T$
12       if $k = s$ or $k = p$ or $k = o$
13          $T := T \setminus \{t\}$
14          $T := T \cup (\mathit{uriToProxy}(s), \mathit{uriToProxy}(p), \mathit{uriToProxy}(o))$
15 for each $(k, v) \in$ uriToProxy
16    $T := T \cup \{(k, \mathit{uriToEqRel}(k), k)\}$
```

The algorithm above is quite intuitive and can be divided into *three* phases. 04–09 removes *all* ERIs from the set of all triples T, guaranteeing at the same time that the mappings are properly updated. Three cases need to be distinguished here (we only describe how *uriToProxy* is modified):[3]

1. both *uriToProxy*(s) **and** *uriToProxy*(o) are **undefined**
   choose s as representative for both s and o: add (s, s) and (o, s).

2. **either** *uriToProxy*(s) **or** *uriToProxy*(o) is **defined**
   say *uriToProxy*(s) is defined: then add $(o, \mathit{uriToProxy}(s))$.

3. both *uriToProxy*(s) **and** *uriToProxy*(o) are **defined**
   union of $[s]$ and $[o]$ is new equivalence class: choose *uriToProxy*(s) as the representative and add $(u, \mathit{uriToProxy}(s))$, for all $u \in [o]$.

10–14 then replaces a URI u by its proxy *uriToProxy*(u), in case $|[u]| > 1$. This is achieved by replacing the triple under consideration by the "proxified" triple. Although equivalence relations are reflexive, we do *not* represent singleton equivalence classes—this would replace a triple by itself. In our implementation, *uriToProxy*(u) is thus undefined for $|[u]| = 1$.

Finally and contrary to what has been said above, 15-16 do add *reflexive proxy triples* $(k\ p\ k)$ to T, where p is one of the OWL equivalence relations. To find

[3]The combination of these three cases is usually called the *Union* operation of a disjoint-set data structure, whereas *proxyToUris* $\circ$ *uriToProxy* refers to the *Find* operation. Thus, such a representation is often called a union-find data structure.

out the “right” p, a third mapping $uriToEqRel : I \mapsto I$ is employed in our implementation that also needs to be updated in the first phase (09). Such a triple will do no harm as it does not interfere with the rules. However, such a triple will become important and will be expanded in the *multiply-out* mode when the repository is being asked “meta” questions, as for instance

```
SELECT ?o WHERE <ltw:Speech_Corpora> <owl:equivalentClass> ?o
```

or

```
SELECT * WHERE ?s <owl:sameAs> ?o
```

3.4 Omitting and Rewriting Rules

Given the last subsection, we can be sure that after a cleanup phase, all ERIs have been removed from a repository. Unfortunately, underspecified ERIs do occur in the entailment rules for the “OWL Horst” dialect of OWL [14], both on the LHS and RHS of a rule. Such ERIs are also used in the new OWL RL profile of OWL 2 [11], but we do focus here on the OWL fragment described in [14] that is used in OWLIM, Jena, or *HFC*.
15 of the 23 P-entailment rules in [14] employ one of the three OWL equivalence relations in predicate position, both in LHS and/or RHS clauses. Given what has been said so far, it will become clear that some of the rules no longer need to be considered, whereas others need to be rewritten. We consider the three cases through rule examples.

3.4.1 Omitting Rules.

A rule, such as **rdfp11** (*HFC* syntax)

```
?u ?p ?v
?u <owl:sameAs> ?u1
?v <owl:sameAs> ?v1
->
?u1 ?p ?v1
```

which copies over information from the LHS to the RHS and which is responsible for the *combinatorial explosion* can now be omitted, since our approach operates on proxies and does **not** consider individuals from the equivalence class of the proxy (with the exception of the proxy itself, of course).

3.4.2 Rewriting Rules: LHS.

Rules which derive new statements without “duplicating” LHS information (see last example), use equivalence relations as additional test patterns (LHS clauses). The following rule signals a problem in case two individuals are regarded to be identical and different at the same time:

```
?x <owl:sameAs> ?y
?x <owl:differentFrom> ?y
->
```

```
?x <rdf:type> <owl:Nothing>
?y <rdf:type> <owl:Nothing>
```

Now, in case no cleanup has been performed after, say

```
<apple-1> <owl:differentFrom> <apple_inc-1>
```

was added to the repository, the above rule will not fire. Even though we find the reflexive proxy triple

```
<apple> <owl:sameAs> <apple>
```

in the repository (see above for the reason why this is so) and although both `<apple-1>` and `<apple_inc-1>` are elements of the equivalence class of `<apple>`, simple symbol matching is not able to derive the two RHS bottom type assignments.

In order to enforce this entailment, we automatically rewrite such rules instead when they are read in. The above rule then becomes

```
?x <owl:differentFrom> ?y
->
?x <rdf:type> <owl:Nothing>
?y <rdf:type> <owl:Nothing>
@test
SameAsTest ?x ?y
```

The important point now is that the matching of (`?x <owl:sameAs> ?y`) is replaced by an external test which is applied *after* LHS clause matching, e.g., `?x` and `?y` are bound, and *before* RHS instantiation. The test simply checks whether the proxies for `?x` and `?y` are the same:

```
$\mathit{SameAsTest}(x, y) \equiv$
  $\textbf{return}\ (\mathit{uriToProxy}(x) = \mathit{uriToProxy}(y))$
```

HFC provides tests for the standard OWL equivalence relations: `EquivalentClassTest`, `EquivalentPropertyTest`, and `SameAsTest`. The set of test can be easily extend to address further equivalence relations.

3.4.3 Rewriting Rules: RHS.

A number of rules from [14] also produce new ERIs through their RHS patterns. **rdfp1**, e.g., deals with functional object properties:

```
?p <rdf:type> <owl:FunctionalProperty>
?p <rdf:type> <owl:ObjectProperty>
?x ?p ?y
?x ?p ?z
->
?y <owl:sameAs> ?z
```

The RHS pattern enforces the values bound to `?y` and `?z` to be part of the same equivalence set. What we would like to achieve here is a rule that generates a reflexive proxy triple (see above), and at the same time unifies the equivalence classes for `?y` and `?z`. In principle, this can be achieved by a test, returning always *true*, having the desired side effect (remember, tests are evaluated after LHS clause matching). However, this would result in an *empty* RHS, not allowed in *HFC*. Fortunately, *HFC* provides so-called *actions*, function returning values which are bound to *RHS-only* variables. Given all this, the following rule is automatically obtained:

```
?p <rdf:type> <owl:FunctionalProperty>
?p <rdf:type> <owl:ObjectProperty>
?x ?p ?y
?x ?p ?z
->
?__actionBinder <owl:sameAs> ?__actionBinder
@action
?__actionBinder = SameAsAction ?y ?z
```

Note that `?__actionBinder` is a brand-new RHS-only variable which binds the proxy for [?y] ∪ [?z] (see also footnote 3):

```
SameAsAction($x$, $y$) ≡
    Union($x$, $y$)
    return Find($x$)
```

Again, *HFC* is shipped with predefined actions for the three OWL equivalence relations.

It is worth noting that even though tests and actions are slightly more expensive than ordinary LHS matching and RHS instantiation, the numbers presented in section 4 immediately show that our approach definitely pays off.

3.5 Querying a Repository

Queries which are posed against a repository might involve URIs that have been substituted in a cleanup swap (see section 3.3). To address this properly, they are rewritten by replacing query URIs in `WHERE` and `FILTER` through their proxies. Even the output of a query might be rewritten in case a user or a program prefers the *multiply-out* expansion mode, instead of the more compact *proxy* mode. Consider the following query which refers to the apple-jobs example from the beginning:

```
SELECT ?o WHERE <apple_inc-1> <owl:sameAs> ?o
```

Assuming that the proxy for `<apple_inc-1>` is `<apple>`, the above query is translated into

```
SELECT ?o WHERE <apple> <owl:sameAs> ?o
```

Depending on the output expansion mode, either the proxy *together* with a pointer to the *proxyToUris* mapping is returned

`?o` ∈ {`<apple>`}

or the whole equivalence set:

`?o` ∈ {`<apple>`, `<apple-1>`, `<apple_computer_inc-1>`, `<apple_inc-1>`, ...}

The size of these two representations differ quite drastically, when more than one variable is employed. Assuming n `SELECT` variables, the *proxy* mode always returns *one* n-tuple $(proxy_1, \ldots, proxy_n)$, whereas the *multiply-out* option yields $|[proxy_1]| \times \cdots \times |[proxy_n]|$ n-tuples—quite a difference, in fact!

4 Measurements

In this section, we evaluate our proposal against ordinary forward chaining that can be found in popular systems, such as OWLIM or Jena. Since the approach requires additional predicates and functions, a high-performance system like OWLIM can not be (directly) extended. Unfortunately, Jena which in principle provides these descriptive means, is *not* able to materialize even drastically-smaller ontologies than the one we have used here as the starting point for our tests. The space and time measurements utilize an ontology that is equipped with a varying number of equivalence relation instances. These data sets are then materialized using *HFC*, both with and without the optimized equivalence relation implementation.

4.1 Ontology, Data Sets, and Setup

The numbers below are computed against the mid-size ontology that backs up the LT-World language portal (see http://www.lt-world.org). The measurements are obtained on a 64bit Intel Core i7 (2.8 GHz), using a 64bit Java 1.6 running in the server mode of the virtual machine. For the experiments, we have configured *HFC* with **two** processor cores, e.g., two entailment rules always run in parallel, if possible.
LT-World incorporates of 1,306 classes, 209 properties, and 17,088 individuals. The unexpanded ABox of LT-World consists of 204,959 RDF triples. LT-World does **not** contain `sameAs` statements, but employs **205** `equivalentClass` and **18** `equivalentProperty` axioms.
The original LT-World ontology, called O_0, is the starting point for our measurements. By adding n randomly-generated (u `<owl:sameAs>` v) triples to O_0, we obtain supersets O_n (u, v URIs from O_0). Needless to say, we make sure that $O_i \subset O_j$ is always the case, for $i < j$.

4.2 Numbers

Without the equivalence class reduction, O_0 results in 548,132 triples. The materialization terminates in 10.2 seconds after 7 iteration steps, taking 712 MB main memory. By switching to the optimized version, things changed to the good side, even already for O_0: 8.8 (+ 0.2) seconds and 511,047 triples. Exactly such numbers, viz., (i) *time to materialize the ontology*, (ii) *time to clean up the ontology* (before and after materialization), (iii) *main memory consumption*, (iv) *number of iteration steps to reach the fixpoint*, and (v) *number of materialized triples* are depicted in the below table.

In order to distinguish between the two versions of *HFC*, we attach a superscript to the data sets: '+' refers to the materialized ontology using the optimized version of *HFC*, whereas '−' identifies the native implementation. As the subscript of O indicates, we perform different measurements for both versions by adding **10**, **100**, **1,000**, **10,000**, and finally **100,000** `sameAs` triples to the original ontology O_0.

ontology	closure[sec]	cleanup[sec]	memory[GB]	#iterations	#triples
O_0^+	8.8	0.2	0.71	7	**511,047**
O_0^-	10.2	0	0.71	7	548,132
O_{10}^+	8.8	0.2	0.71	7	**510,932**
O_{10}^-	10.6	0	0.71	7	548,520
O_{100}^+	8.7	0.2	0.71	7	**509,891**
O_{100}^-	10.9	0	0.72	7	554,027
$O_{1,000}^+$	8.6	0.2	0.71	7	**499,024**
$O_{1,000}^-$	16.9	0	0.89	7	704,149
$O_{10,000}^+$	6.4	0.5	0.67	9	**328,221**
$O_{10,000}^-$	——[4]	0	——[5]	——[6]	——[7]
$O_{100,000}^+$	1.6	0.9	0.49	5	**118,104**
$O_{100,000}^-$	——[8]	——[8]	——[8]	——[8]	——[8]

As one can see from the above table, we were **not** able to finish the experiment for the **native** approach utilizing only **10,000** `sameAs` triples. The reason why this experiment ended so badly came from the fact that the synthetically generated `sameAs` statements were filled in subject and object positions with one of the 17,088 URIs from O_0. With 10,000 `sameAs` ERIs, it was the case that these ERIs heavily interacted with one another (as one expects), so that the rule for transitive properties together with the expensive copy rule **rdfp11** blew up the materialization process.
Contrary to the native approach, the **optimized** version not even *avoids* the combinatorial blowup attributed to equivalence relations, but can also *reduce* the overall number of triples in case more `sameAs` triples are added (see bold numbers above). Note that the materialization of $O_{100,000}^+$ even contains less triples than the unexpanded original ontology O_0!
This seems to be strange at first sight. Why is this so? Consider the following simple example. Assume that our repository contains both

```
<s1> <p> <o>
```

and

```
<s2> <p> <o>
```

By adding

```
<s1> <owl:sameAs> <s2>
```

[4]Closure computation was stopped after 5 minutes.

[5]15 GB main memory was exceeded then.

[6]Iteration **4** was still ongoing and got stuck in the expensive copy rule **rdfp11** for subject and object position.

[7]Unknown, but **rdfp11** had already produced more than **2,900,000** new triples in iteration **3**, when materializing $O_{10,000}^-$.

[8]Not tried, since $O_{10,000}^-$ already failed.

the above two statements become the same and the cardinality of the set of triples is reduced by one.

5 Summary, Remarks, and Comparison

We hope to have shown that an explicit equivalence class reduction implementation for all three OWL equivalence relations can have a drastic effect on both the memory footprint, but also on the materialization time of a practical triple repository. The approach has been implemented in the forward chainer *HFC* and has been profiled with a varying number of `sameAs` statements for the mid-size ontology LT-World. The approach is distinguished by a separate cleaning-up phase and a specific rule-rewriting mechanism, realized in *HFC*. Not even materialization benefits from this approach, but also query results can be drastically more compact, when compared to the naïve approach, as section 3.5 has shown.

In footnote 1, we indicated that it might be useful to have a further property characteristics, say `EquivalenceRelation`, in order to let other properties participate in this optimization. We have also indicated that this optimization carries over to and is already implemented for arbitrary *tuples* that can be used to directly encode general n-ary relations.

To the best of our knowledge, we are not aware of "real" descriptions, presenting the efficient *implementation* of all three equivalence relations in OWL in such a detail, as done here. Some of the popular RDF stores do not look into reasoning very much, focussing more on efficient representation and querying (e.g., Virtuoso), thus do not handle such ERIs. Other forward chaining engines either do not address equivalence class reduction (e.g., Jena) or claim to implement certain optimizations in their commercial enterprise engines (e.g., BigOWLIM), probably not wanting to reveal their "secrets". The treatment of ERIs need *not* be implemented within the reasoner, but can also be sourced out in a separate module, as we have described in [8]. However, such a strategy is less efficient and not optimal than the technique described here.

Two papers are worth to mention here which address `sameAs` statements in their systems, viz., Owlgres [13] and an extension of Oracle 11g [5]. Contrary to the system described here, both systems rely on a DB back end for efficiently storing and querying triples. Unfortunately, nothing is said how both systems do treat rules which involve `sameAs` statements directly or indirectly as we have described in Section 3.4. Given the two papers, it seems that both systems only rely on cleaning-up phases as we have explained in Section 3.3. Both systems as well as ours use a standard union-find data structure to implement equivalence classes as is common in computer science [1].

This paper presents *one way* of implementing an equivalence class reduction and is clearly applicable to Jena, since test and actions can also be found in this engine. Other systems, such as SwiftOWLIM, which do not provide custom functionality can in principle also be equipped with such a kind of optimization, assuming that the engine is sensitive to the "critical" rules, adds and executes "hard-wired" functionality (i.e., programming code) during the interpretation phase of the rules.

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge, MA (1990)

[2] Frank, A., Krieger, H.U., Xu, F., Uszkoreit, H., Crysmann, B., Schäfer, U.: Question answering from structured knowledge sources. Journal of Applied Logics, Special Issue on Questions and Answers: Theoretical and Applied Perspectives 5(1), 20–48 (2007)

[3] Kiefer, B., Krieger, H.U.: A context-free approximation of Head-Driven Phrase Structure Grammar. In: Proceedings of the 6th International Workshop on Parsing Technologies, IWPT2000. pp. 135–146 (2000)

[4] Kiryakov, A., Ognyanov, D., Manov, D.: OWLIM – a pragmatic semantic repository for OWL. In: Proceedings of the International Workshop on Scalable Semantic Web Knowledge Base Systems. pp. 182–192 (2005)

[5] Kolovski, V., Wu, Z., Eadon, G.: Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In: Proceedings of the 9th International Semantic Web Conference (2010)

[6] Krieger, H.U.: A general methodology for equipping ontologies with time. In: Proceedings LREC 2010 (2010)

[7] Krieger, H.U., Kiefer, B., Declerck, T.: A framework for temporal representation and reasoning in business intelligence applications. In: AAAI 2008 Spring Symposium on *AI Meets Business Rules and Process Management*. pp. 59–70. AAAI (2008)

[8] Krieger, H.U., Kiefer, B., Declerck, T.: A hybrid reasoning architecture for business intelligence applications. In: 8th International Conference on Hybrid Intelligent Systems, HIS-2008. pp. 843–848. IEEE (2008)

[9] Krieger, H.U., Kruijff, G.J.M.: Combining uncertainty and description logic rule-based reasoning in situation-aware robots. In: Proceedings of the AAAI 2011 Spring Symposium "Logical Formalizations of Commonsense Reasoning" (2011)

[10] Manola, F., Miller, E.: RDF primer. Tech. rep., W3C (2004)

[11] Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 web ontology language profiles. Tech. rep., W3C (2009)

[12] Reynolds, D.: Jena rules tutorial. In: Jena User Conference (2006), PowerPoint presentation

[13] Stocker, M., Smith, M.: Owlgres: A scalable OWL reasoner. In: Proceedings of the 5th OWLED Workshop "OWL Experiences and Directions" (2008)

[14] ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Journal of Web Semantics 3, 79–115 (2005)

-11-01

ch Report