KR DL PROJECT REPORT GROUP 21

Vrije Universiteit Amsterdam
Knowledge Representation
Description Logics
Group 21

ABSTRACT

For this project on $\mathcal{EL}$, an ontology on burgers was constructed, a reasoner to reason with $\mathcal{EL}$ was constructed, and an experiment was done to test the computational efficiency of different permutations of rule applications in the algorithm. Because of mistakes, the experiment did not succeed as planned, and instead a quick subset of the experiment was performed, which outputted no statistically analysable data, but did suggest that rule application permutations can play a role in the computational efficiency of $\mathcal{EL}$ reasoners.

***K*eywords** Description Logic · Knowledge Representation · Ontology · $\mathcal{EL}$ Reasoner · Burgers

1 Introduction

This report takes a flavourful journey into the ontology of burgers, aiming to uncover the intricate layers of meaning and significance woven into the seemingly simple combination of bun, patty, and toppings. Subsequently, the implementation of an $\mathcal{EL}$ Reasoner is carried out to verify all subsumers within the chosen domain. The report is divided in three main parts: The Ontology, Reasoner Implementation, and The Creative Part.

Firstly, why burgers? The prevalence of burgers in various cuisines and the adaptability of their components make them an ideal subject for study. Also, the straightforward structure of a burger—consisting of a bun, patty, and toppings—renders it both accessible and nuanced for ontological examination. An Ontology is a meticulously structured framework for the organisation knowledge by defining the essential categories and elucidating their interrelations. Ontologies allow us to define a set of axioms that capture the essential characteristics of a domain by specifying the relationships and constraints that govern the entities within the ontology.

Secondly, in ontology development, $\mathcal{EL}$ Reasoners are employed to efficiently manage and reason about complex relationships and concepts within a domain by striking a balance between expressive power and computational efficiency.

Lastly, for the creative part, ontological reasoners face run time efficiency issues with hardly any defined mechanism which promises significant speedups. Therefore, we suggest conducting research to assess the variations in run time efficiency among different permutations of algorithm rule application orders.

2 Ontology Description

2.1 Overview

The ontology we made for this project is considered around burgers. It is aimed at providing any restaurant with three points of information: what type of menu item it is, generally, the allergens and dietary constrictions contained in each menu item, and if the menu item is spicy. It does these things based of constructing burgers out of a set of ingredients, for which information is provided.

The first of these has to do with what type of customer would order the item. There are three of these, filling, healthy, and cheeseburger. Filling burgers, here, are the burgers one might order if one is coming to the restaurant because they are hungry and healthy burgers are the burgers that contain some amount of vegetables. The two of these together might indicate whether someone would order the item if they are planning to have a full meal at the restaurant. The last, cheeseburger, is there simply as a reminder that certain customers will either not or only order burgers with cheese, and that this should be considered when making a menu.

The allergens and dietary constrictions consist of two parts: allergens, and pescatarian / vegetarian / vegan. For the allergens, the system is designed to indicate whether something contains traces or ingredient amounts of an allergen. There are indicated with the [Allergen]UnsafeTraces and [Allergen]Unsafe classes respectively. Pescatarian / Vegetarian / Vegan are instead all classes that a menu item can be a subclass of, showing if the menu item satisfies the requirement to fit within those dietary restrictions.

The last, spiciness, is simply a label that states if a menu item is spicy, or, to be more precise, if the burger contains any spicy ingredients.

For all of these, the reason that they exist as named classes, rather than just having the extension be there, is that the reasoner checks relationships between named classes. Though '$\exists containsTracesOf.FishAllergen$' would still hold true for the appropriate items without these named classes, menu items would not be classified as such by the reasoner without explicit query.

The main focus, after all of this, are the named burgers. These are the actual menu items. In the ontology right now, there are what can only be classified as examples, a set of burger menu items that should show the capabilities of the reasoner. One of these is there to show one of its limitations: the HotIsHealthy burger. Because Jalapenos are vegetables, a burger that includes them is automatically healthy, and, of course, spicy.

2.2 Technical Explanation

Technically, most of the features are achieved by subclassing transitive properties.

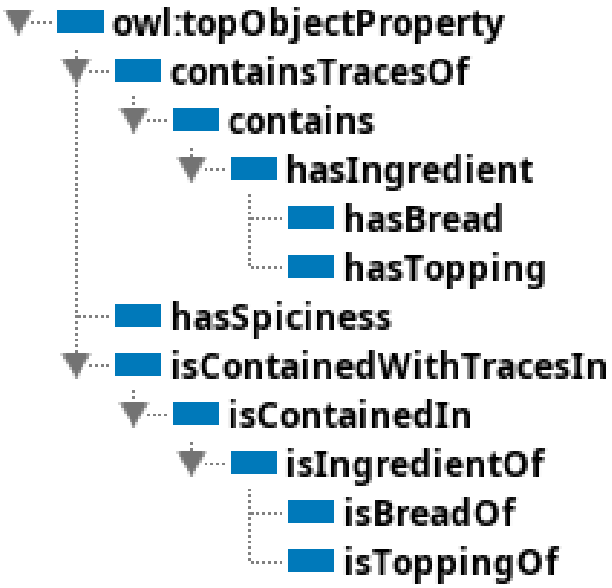

Figure 1: Ontology Tree

The burgers all have ingredients that are a part of this ontology, done with the subclasses of hasIngredient, For these ingredients, it is specified if they are spicy (with the hasSpiciness property), if they are pescatarian/vegetarian/vegan (by subclassing), and if they contain (traces of) any allergens (via contains/containsTracesOf). This is done as such:

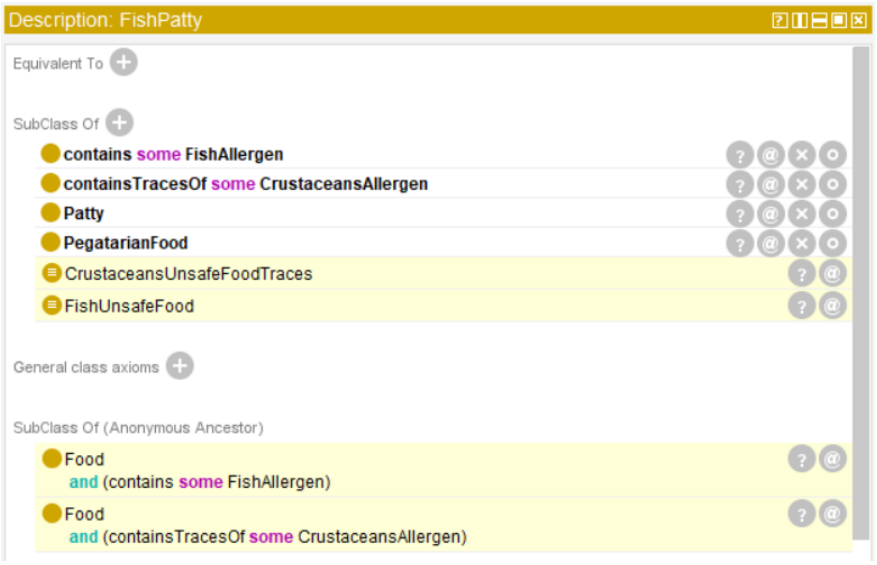

Figure 2: Fish Patty Description

After this, the named burgers can be constructed as the following, by simply stating they have exactly their ingredients:

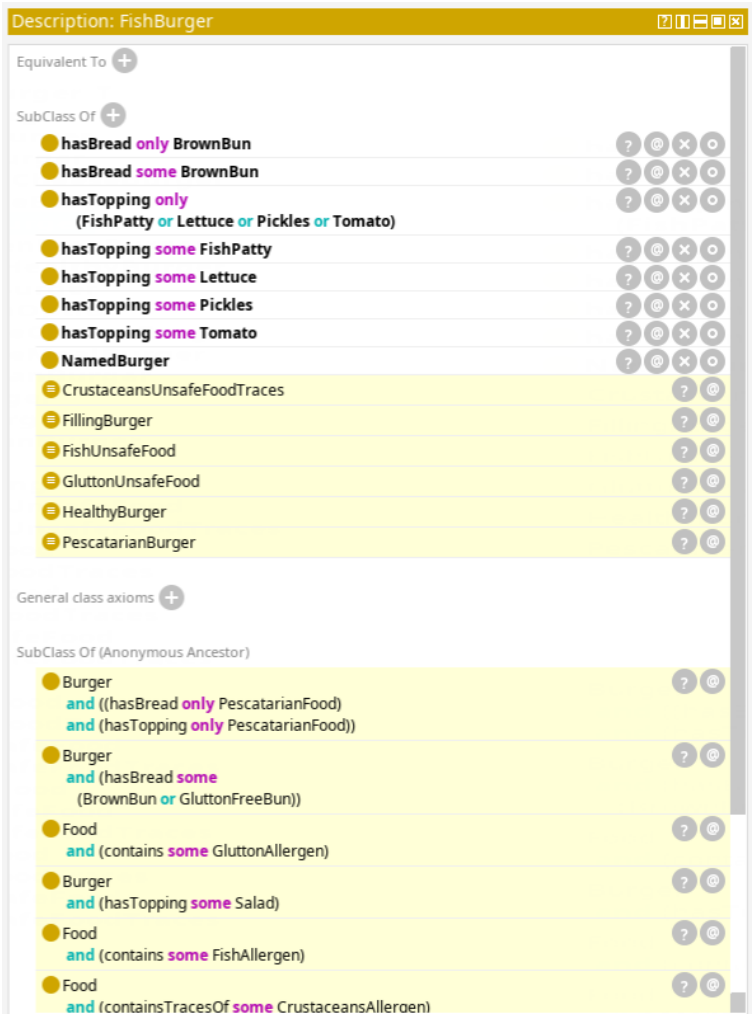

Figure 3: Fish Burger Description

Because of this, the SpicyBurger class menu items can be found to be a subclass of can simply be equivalent to:

$Burger \sqcap (\exists hasBread.(BurgerBread \sqcap \exists hasSpiciness.Hot)) \sqcup (\exists hasTopping.(BurgerTopping \sqcap \exists hasSpiciness.Hot))$

The PescatarianBurger can simply be:

$Burger \sqcap \forall hasBread.PescatarianFood \sqcap \forall hasTopping.PescatarianFood$

and the allergens can all be constructed like:

$Food \sqcap \exists contains.PeanutAllergen$

Because all properties are transitive, and because they subclass each other, a named burger will be found to contain glutenAllergens if the burger hasBread some ingredient that contains some gluten.

Moreover, because of the subclassing, the reasoner can figure out on itself that everything that is vegan is also vegetarian, and something that contains some allergens also contains traces of that allergen.

3 Reasoner Implementation

3.1 $\mathcal{EL}$ Description

To reason about our defined ontology, we implement a basic $\mathcal{EL}$ reasoner in Python. The library *dl4python* is employed, creating a Java gateway server within a JVM, allowing the code to interface with the OWL API, which is only available for Java. Through this library, the Ontology can be parsed and reasoned with, allowing us to make new inferences.

The $\mathcal{EL}$ algorithm is defined to decide whether one concept is subsumed by another, formally $\mathcal{O} \models C_0 \sqsubseteq D_0$, where it must be decided whether C_0 subsumes D_0. This can also be referred to as *general concept inclusion* (GCI).

3.2 Algorithm Definition

The algorithm first picks an initial element d_0, assigned with initial concept C_0, then exhaustively applies a set of rules which assign new concepts from the input to d_0. The algorithm accepts that $C_0 \sqsubseteq D_0$ if D_0 gets assigned to d_0.

The set of rules are defined as such:

0. assign T to any individual
1. if d has $C \sqcap D$ assigned, assign also C and D
2. if d has C and D assigned, assign also $C \sqcap D$
3. if d has $\exists r.C$ assigned, make e the r-successor of d, if there is an initial concept e with C assigned, else add a new r-successor to d and assign it concept C
4. if d has an r-successor with C assigned, add $\exists r.C$ to d
5. if d has C assigned and $C_0 \sqsubseteq D_0 \in \mathcal{T}$, D to d

These rules are implemented simply in Python. An $\mathcal{EL}$ *Reasoner* class is defined for easy importing of the reasoner into other files, which is especially important for running our experiment, addressed in the next section. The class' *init* method initialises all of the basics needed to run the reasoner, opening the Java gateway, importing the ontology, and converting all of the conjunctions to a binary representation.

The class also defines five individual functions which can apply individual rules to the initial element d_0, as well as one function to apply them iteratively and exhaustively. Finally, a function *check_subsumption* is defined, which uses a dictionary containing individuals as keys and their set of assigned concepts as values to test whether one class is subsumed by another.

Technically, $\mathcal{EL}$ has another to deal with equivalence. This was left out of the present experiment.

4 Creative Part

As previously mentioned, the $\mathcal{EL}$ algorithm has five rules, which can be applied in any order to d_0, as long as only concepts from the input are assigned. To this end, we propose research to determine differences of efficiency between various permutations of the algorithm rule application order.

Runtime efficiency is one of the leading issues facing ontological reasoners, with few known methods of guaranteeing consistent speedups. To illustrate this point, we can show trivially that

reasoning with description logics (DLs) is harder than in propositional logic, due to the potential for infinitely many interpretations which involve various domain elements. Full reasoning with DLs can be done with $\mathcal{ALC}$ reasoners, but they may require exponential time in the size of the ontology to find subsumers.

As these these $\mathcal{ALC}$ reasoners are both inefficient and more difficult to implement, we stick to using and attempting to improve upon an $\mathcal{EL}$ reasoner, which should have a worst-case efficiency in n^2 polynomial time, where n is the size of the ontology. Our experiment involves firstly implementing this $\mathcal{EL}$ reasoner in Python, which we have described above, and then testing all possible permutations of rule application order to determine the most efficient reasoning method.

As being able to efficiently reason about very large ontologies is important across a range of fields, our research question attempts to address this simple potential method of gaining considerable speedup time without complicated implementation.

Unfortunately, a programming mistake meant that the code that was supposed to run overnight to test all permutations on numerous ontologies did not do as it was supposed to, and no results were obtained until Saturday morning, at which point there was no time to run all permutations, or on all sorts of different ontologies. Instead, our own ontology was used, testing for all subsumers of the class "Salad" in six orders which can be constructed by splitting the list in half, and then getting all ordered permutations of that, chosen simply because something had to be cut out, and this seemed like the most spread out way to do that. Because this was ran on only one ontology, testing the subsumers of only one class, no statistics can be done on the resulting data. Instead, this experiment should be seen as the quickest form of a pilot study.

5 Results

The results are as follows, with the rule permutations ordered from least to most efficient:

Figure 4: Rule Order vs. Run Time

The experiment was run on Debian Linux with python 3.11, running on a machine that had quite some other stuff running in the background. The time differences are large enough that we hope this did not influence the results too much, but this is only hope.

As we cannot do any statistics on this data, we cannot show one way or another if these are significant, and how much the effect matters. However, as a pilot, we do think the data suggests that some optimisation of the algorithm can be found in running the rules in different orders.

6 Conclusion

Unfortunately, as we were unable to run our experiment on the entire set of 120 possible permutations, or repeat multiple iterations of the experiment to test its reproduce ability, we cannot make strong conclusions as of this moment. There does appear to be definite differences in the run times of different permutations, but, as previously mentioned, without more rigorous evaluation or statistical analysis, this cannot be verified. We do believe, however, that these results show promise, and suggest that much future work could be done on this topic.

7 Limitations and Future Work

Due to our limited available computing power as well as a lack of processing time before the project deadline, we did not manage to test our hypothesis or methodology as rigorously as we would have liked. To verify our results, testing our permutation method on various ontologies would be essential, to test whether it can generalise well.

It would also be pertinent to run each test multiple times and take an average of the time elapsed for each run, as this would confirm better our results and allow for any inconsistencies. Furthermore, implementing an equivalence rule would have benefited the experiment, as, at that point, our reasoner could have been compared with a publicly available one to assert all outputs are equivalent.

Lastly, in terms of future work, extending our reasoning capabilities to include disjunctions in our ontology, as well as testing whether the permutations still have an impact on run time with an extended reasoner would be an interesting topic to address.