Spicy Formalisms

Constructing a Reasoning System for Curry Restaurants using Description Logic

Alexandra Genis[1], Mikel G. W. Blom[1], and Daimy van Loo[1]

Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam

Abstract. Curry is a popular dish in many countries and is unique in its versatility. To facilitate the customer experience at curry restaurants we develop an intelligent reasoning system employing Description Logic formalisms. Using this system allows to identify menu items that suit the individual preferences and dietary habits of guests.

Keywords: Curry · Ontologies · Description Logic · Knowledge Representation

1 Introduction

Curry is a popular food around the world and found in various restaurants and chains in many cities. Often restaurant menus feature a large variety of curries, which can make it difficult for guests to have an overview of dishes fitting their preferences and dietary restrictions. To enhance the customer experience of curry restaurants, we propose the development of an intelligent reasoning system, which can be queried to meaningfully find items on the restaurant menu while considering dietary restrictions and preferences.

The following report describes the construction of such an intelligent reasoning system to support curry restaurants.

Description Logic (DL) appears to be a promising formalism for system construction, consisting of an ontology and reasoning algorithm (reasoner), which facilitate the storing and querying of information respectively. The present study implements an ontology broadly leaning on but not exclusive to Attributive Concept Language with Complements (ALC) and an Efficient Reasoning (EL) reasoner.

2 Methods

The main components of an intelligent reasoning system are an ontology and a reasoner. The ontology contains a collection of all concepts and relationships within the domain of interest while the reasoner is an algorithm querying the ontology to find out whether a concept or relation between two concepts is entailed by the ontology, so can be assumed to be true in our domain of interest.

The goal of the present study is to create a curry ontology, which can be queried by a reasoner, for example to find out whether a given dish has certain qualities, such as being vegetarian or being spicy.

2.1 Implementation Details

Curries and other dishes utilized in the ontology were partially inspired by [1]. The ontology was implemented and queried in Protégé [4] through the HermiT reasoner in addition to an EL reasoner, which was constructed in python [5] with use of the dl4python library [2].

2.2 Ontology

The curry ontology constructed for the present study was inspired by the pizza ontology [3] and contains 260 axioms, 70 classes, eight roles, and four individuals. In addition, the ontology contains 107 subclasses, 11 equivalences and 14 disjoint classes.

The terms classes and concepts are used interchangeably, similarly to roles and object properties, as these terms describe the same entities in DL and OWL terminology.

The proposed ontology is consistent, meaning that it has a model, does not contain contradictory statements, and does not entail all axioms. Further, it is also coherent, as all concepts are satisfiable.

Classes Concepts are nested in the following hierarchical structure through the use of concept inclusion: initially we construct the root class $owl : Thing$, which contains the *Domain* of all dishes included in the ontology. The *Domain* consists of two disjoint Subclasses, namely $Country$ and $Food$. $Country$ contains a set of nominals representing the countries of origin, which is intuitive as the four countries included are instances of the concept $Country$ and not separate concepts in themselves.

$Food$ forms an umbrella class for all menu items and their ingredients. $Food$ further contains five disjoint subclasses, namely $Bread$, $Curry$, $CurryBase$, $CurryIngredient$, and $Rice$. Of these classes each again contains subclasses denoting their respective items such as $Naan$ nested in $Bread$ or $Broth$ nested in $CurryBase$.

Subclasses of $Curry$ can be concepts of different curry types, such as $VegetarianCurry$. One special class in $Curry$ is $NamedCurry$, which contains further subclasses of iconic curries such as $ButterChicken$ or $MassamanCurry$. Another special case is presented by $ComplexCurry$, which is defined by a number restriction, as it needs to contain a minimum of five ingredients to be satisfied.

The subclasses of a given class are meaningfully disjoint if they are mutually exclusive. For example an individual assigned class $AnimalIngredient$ cannot be a $VegetableIngredient$ or $Salt$ and an instance of $Rice$ cannot be $BasmatiRice$

and $JasmineRice$ simultaneously. Likewise, an instance of $KatsuCurry$ cannot be an instance of $MassamanCurry$ at the same time.

Next to the $Domain$, we also provide the class $ValuePartition$, which contains information on the spiciness of a dish in form of the subclasses $Mild$, $Medium$, Hot, and $Extreme$. The status of $ValuePartition$ implies that only other classes but not individuals can be values in the set of this class. This makes sense as spiciness is a category we want to assign to certain menu dishes, not just individual plates. $Domain$ and $ValuePartition$ are disjoint.

Roles / Object Properties Relationships between the different food items are described by roles / object properties, of which eight are included in the present ontology. An example is the role $hasCountryOfOrigin$, which connects an individual with the concept $Country$ to an individual assigned a $Food$ concept.

Some relationships can also be the inverse of each other, such as $hasCurryBase$ and $isCurryBase$. $HasCurryBase$ connects concepts in the superclass $Curry$ with concepts from $CurryBase$. $IsBaseOf$, on the other hand, connects the same concepts but in reverse order, so connections have the opposite direction.

The present ontology also contains transitive relationships such as the roles $hasIngredient$ and $isIngredientOf$. These roles are transitive as there are foods which are ingredients to other ingredients. For example, some spices might be ingredients in $GaramMasala$, which in itself is an ingredient in some curries. Accordingly, this role spans multiple nested layers of the concept hierarchy.

ALC and SROIQ(D) Language Use There are multiple instances described in the section above where the curry ontology goes beyond the expressivity of ALC and which are rather part of the more expressive language SROIQ(D). Examples are the use of nominals, transitive and inverse roles, number restrictions, and role hierarchies.

Nevertheless, the present ontology takes full advantage of the existing expressivity of ALC by utilizing all available concept constructors, namely $\top$, $\bot$, $\neg$, $\sqcap$, $\sqcup$, $\exists$, and $\forall$.

While $\top$ is implied in the root node which encompasses everything that exists in the ontology, $\bot$ is implied in disjoint concepts, as:

$$(VegetarianCurry \sqcap NonVegetarianCurry) \sqsubseteq \bot$$

A curry cannot be vegetarian and not vegetarian simultaneously, thus this concept does not exist and these concepts are disjoint. Further, negations are present in concepts such as $VegetarianCurry$ and $NonVegetarianCurry$ as the former negates animal ingredients and the latter is a negation of the former. Conjunctions $\sqcap$, disjunctions $\sqcup$ and existential restrictions $\exists$ can be found in examples such as:

$$SpicyIngredient \equiv CurryIngredient \sqcap (\exists hasSpiciness.(Extreme \sqcup Hot))$$

Here, a $SpicyIngredient$ is satisfied only if it is a $CurryIngredient$ and has either Hot or $Extreme$ spiciness. Finally, an example for value restrictions $\forall$ is:

$$MassamanCurry \sqsubseteq \forall hasCurryIngredient.(Chicken \sqcup Onion \sqcup Potato)$$

This implies that $MassamanCurry$ can only have three curry ingredients, namely $Chicken$, $Onion$, $Potato$, and any of their combinations. The above examples also illustrate the presence of complex concepts in the ontology, which consist of combinations and interactions between other concepts.

2.3 EL Reasoner

To explore possible inferences using the curry ontology we construct an EL reasoner. This reasoning algorithm is designed to assess subsumption $C \sqsubseteq D$ of a given concept C by a given concept D. The EL reasoner does so through the use of a canonical model $\mathcal{O}$ of the ontology, in which an individual d is constructed and given the interpretation $C^{\mathcal{I}}$. Further, all axioms provided by the ontology, which need to be met to satisfy C are applied to d, and any other individual, while respecting a set of six inference rules.

First, d is assigned $\top$ meaning that it exists as a prerequisite for all further inferences. Second, through the $\sqcap$-rule 1, if d is assigned $C \sqcap D$ in the ontology, it receives separate assignments of C and D. Third, the $\sqcap$-rule 2 is a reverse of $\sqcap$-rule 1 and demands that if d is assigned both C and D it also needs an assignment of $C \sqcap D$. Fourth, the $\exists$-rule 1 demands that if d is assigned with $\exists r.(C)$, and there is no r-successor present in C, this successor must be created. Fifth, the $\exists$-rule 2 implies that if d has an r-successor in C, it must also be assigned the axiom $\exists r.(C)$. Finally, the $\sqsubseteq$-rule postulates that if d is assigned C and $C \sqsubseteq D \in T$, D must also be assigned to d.

If at the end of this process D has indeed been assigned to d and no more rules can be applied, $C \sqsubseteq D$ is true. This process can be used to obtain all subsumers. Within the curry ontology, this reasoning can assess which categories a given curry or ingredient is part of. For more insight into the algorithm a pseudocode is provided.

Challenges for EL Reasoning While EL reasoning is generally very efficient as inferences are possible in polynomial time, this efficiency goes hand in hand with several limitations.

Most prominently, the EL reasoning algorithm omits the use of bottom $\bot$, disjunctions $\sqcup$, negations $\neg$, and value restrictions $\forall$. Axioms including these constructors are simply ignored by the reasoner. This being the case, EL reasoning is quite limited to inferences such as instance checking, classification, and materialization.

Additionally, there are two major challenges tied to the standard EL algorithm. First, the standard $\sqcap$-rule 2 tends to generate new concepts. If an individual a has concepts B and C assigned to it, the algorithm will initially

Algorithm 1 EL Algorithm to Obtain all Subsumers

```
1: function EL
2:     add initial individual d and assign concept C (subsumee)
3:     while interpretation changed do
4:         for individual i in interpretation do
5:             check whether i is blocked
6:             if i not blocked then
7:                 add ⊤ to i                                   ▷ apply ⊤-rule
8:                 if i: C ∩ D then
9:                     assign i : C and i : D                   ▷ apply ∩-rule 1
10:                end if
11:                if i : C and i : D and C ∩ D ∈ concepts then
12:                    assign i : C ∩ D                         ▷ apply ∩-rule 2
13:                end if
14:                if i : ∃r.C but no r.C̲ successor assigned then
15:                    if some individual j : C̲ exists then     ▷ apply ∃-rule 1
16:                        make j r successor of i
17:                    else
18:                        add new r successor h and assign h : C̲ to it
19:                    end if
20:                end if
21:                if i has r successor e : C and ∃r.C ∈ concepts  then
22:                    assign i : ∃r.C                          ▷ apply ∃-rule 2
23:                end if
24:                if i : C and C ⊆ D ∈ T then
25:                    assign i : D                             ▷ apply ⊆-rule
26:                end if
27:            end if
28:        end for
29:    end while
30:    subsumers = emptylist
31:    for concept in name_concepts do
32:        if d : concept then
33:            add concept to subsumers
34:        end if
35:    end for
36:    return subsumers
37: end function
```

create a concept $B \sqcap C$ and assign it to a. Yet, then a has three concepts, and according to the rule, the algorithm then has to proceed with $B \sqcap B \sqcap C$ and $C \sqcap B \sqcap C$, etc. As this process would pose a challenge for termination of the algorithm, the proposed implementation restricts $\sqcap$-rule 2 to only apply to input concepts. Thus a can only be assigned intersections if they occur in the input of the reasoner.

Second, the $\exists$-rule 1 as described in the standard implementation of the EL could lead to infinite addition of new individuals. If there is a concept $C \sqsubseteq \exists r.C$,

this rule will add a new individual with role r and concept C to every individual with concept C. To resolve this issue, we reuse individuals in the $\exists$-rule 1, because we assess for existing individuals with the relevant initial concept C. Only if such individuals do not exist, a new individual is added.

2.4 EL and HermiT System Comparisons

We compare the constructed EL reasoner with the HermiT reasoner provided in Protégé. These algorithms differ in the way they handle ontology expressivity, with EL being less flexible than HermiT, as the latter is able to make use of the full expressivity of the curry ontology. These differences inherently affect the queries and concepts which can be used with either of them and the results they produce.

3 Results and Analysis

This section presents several application cases for the curry ontology and compares EL and HermiT reasoning systems.

3.1 EL

The EL reasoner allows for certain inferences using the curry ontology. For example, an inference of *Coriander* returns all classes it is a subclass of, such as *Food* and *CurryIngredients*.

It is also possible to infer ingredients of dishes, by using for example *Chicken Curry*, which returns all curries containing *Chicken*. The complex concept *ChickenCurry* is synonymous with $Curry \sqcap \exists hasCurryIngredient.(Chicken)$. Concepts such as *ChickenCurry* are useful, as the implementation of the EL algorithm used in the present study doesn't allow to query complex concepts unless they are expressed as a name concept in the ontology.

Additionally, as mentioned in the EL reasoner limitations, certain concept constructors, such as $\neg$ are excluded from the reasoning process. This makes it challenging to query for concepts such as $VegetarianCurry$ as they contain the exclusion of certain ingredients. To resolve this problem it would be necessary to place all vegetarian curries in the superclass $VegetarianCurry$ manually. However, the aim of the curry ontology is to allow for an automated inference of $VegetarianCurry$ based on its ingredients.

Further limitations concern the use of transitiveness and value restrictions, which are not possible with EL.

3.2 HermiT

The functionality of the HermiT reasoner likewise allows to perform simple inferences such as inquiring whether a given curry is vegetarian or spicy. Additionally, the flexibility of HermiT provides an opportunity to perform more interesting

and complex inferences with the curry ontology, which involve the interaction between multiple concepts and axioms.

For example, it is possible to search for a certain type of curry which also contains a certain ingredient, such as a vegetarian curry containing Paneer cheese. This query can look like the following:

$$VegeratianCurry \sqcap PaneerCurry$$

yielding the subsumee $ButterPaneer$. $PaneerCurry$ here is a concept representing the role $\exists hasIngredient.(Paneer)$ and aids the convenience of querying.

We can also look for curries with a certain base and a certain ingredient such as:

$$CreamBasedCurry \sqcap \exists hasCurryIngredient.(ChilliPowder)$$

also yielding $ButterPaneer$. Apart from the possibility to make queries for an intersection of concepts, this also demonstrates how the same concept can be the result of many different queries due to the interactions between axioms.

Transitiveness allows for other interesting inferences. For example the same $ButterPaneer$ contains $GaramMasala$, which in turn contains $Coriander$. A query for

$$Curry \sqcap \exists hasIngredient.(Coriander)$$

thus also yields $ButterPaneer$ without $Coriander$ being explicitly mentioned as an ingredient.

A final interesting case is the combination of

$$ComplexCurry \sqcap SpicyCurry$$

which allows to infer whether a curry has more than five ingredients and is spicy, so has Hot or $Extreme$ spiciness. Thus, the reasoner needs to count and evaluate the number of concepts in the curry which are ingredients and also whether one of these ingredients satisfies the criteria of $SpicyCurry$. This query returns $Vindaloo$ as a subclass.

The queries above demonstrate that while EL reasoning is more efficient, its limitations also need to be considered in ontology construction and can prevent the opportunity for certain inferences. Accordingly, for smaller sized ontologies such as the curry ontology, it is beneficial to consider the use of more expressive reasoners such as HermiT despite of higher complexity.

4 Conclusion

The present research investigated the construction of a reasoning system to enhance the customer experience at curry restaurants. We developed a curry ontology to allow queries of restaurant menus. Additionally we compared an EL and HermiT reasoning system applied to this ontology. Based on this study the use of HermiT appears more effective to query the menu for guest preferences and dietary requirements. We conclude that the expressivity of an ontology needs to be supported by the reasoning system to be fully usable.

References

1. Dan Toombs: The curry guy bible: Recreate Over 200 Indian Restaurant and Takeaway Classics at Home. 1st edn. Quadrille Publishing Ltd, London (2020)
2. Koopmann P.: dl4python (n.d.)
3. Horridge M.: Pizza Ontology, retrieved November 2024, https://github.com/owlcs/pizza-ontology/blob/master/pizza.owl
4. Stanford University: Protégé: A free, open-source ontology editor and framework for building intelligent systems. Version 5.6.4, https://protege.stanford.edu/
5. Python Software Foundation: Python: A programming language that lets you work quickly and integrate systems more effectively. Version 3.11, retrieved November 2024, https://www.python.org/.