VRIJE
UNIVERSITEIT
BRUSSEL

THE LAST RECIPE

Open Information Systems
Project Report

Khaïm Berkane
(khaim.berkane@vub.be - 0555007),
Aaron Lippeveldts
(aaron.joos.lippeveldts@vub.be - 0536407),
Kelvin Schoofs
(kelvin.schoofs@vub.be - 0557935),
Stijn Vissers
(stijn.luc.vissers@vub.be - 0567854)

2019 - 2020

Science & Bio-engineering Sciences

Contents

Abstract

The last recipe is our application for the Open Information Systems project.

This report contains the thought process and results of the creation of our system. The project started with the creation of a conceptual schema. This paper will discuss the limitations of our design and the solutions to them. The ontologies have been created by using the Protégé tool[3]. A visualisation of these ontologies has also been generated using the WebVOWL[4] visualisation tool. SHACL constraints were also implemented to check our ontology, this can be validated in Protégé using a plugin[2]. Furthermore, this paper will demonstrate how our ontology can be used in a non-trivial matter. This paper shall conclude with the way our group worked together to achieve this result. Our final result can be found in on online Git repository[1].

1 Introduction

The last recipe is our implementation of an ontology for recipes. Emphasizing tools and ingredients, it allows users to query recipes based on those and even obtain alternatives. Ingredients belong to different categories, this allows us to classify the recipe they are used in to a particular diet. Various information about Ingredients can be encoded, such as calories or prices. Our design started with dividing a recipe in various steps. To support our features we had to put a number of restrictions in place, which clients of our system have to adhere.

2 Conceptual Schema

To support more fine-grained queries we decided that a recipe would consist of several cooking steps. A step, besides a human-readable description of the action to execute, references a tool, an ingredient, and the time required to execute this step. This makes it easy for filtering recipes with equipment or time constraints.

A cooking step has at most one ingredient. For example mixing the ingredients for French crepes could be done in 4 steps as follow:

- Pour 250g of *flour* in a *bowl*
- Add a pinch of *Salt* and make a well in the center
- Add 4 large *eggs*
- Add 50g of *melted butter*

This could also be done in 5 steps by splitting the second in two: “make a well in the center” would have no ingredients nor use any tool. The same holds for tools used in the recipe. If we had a step “deep fry in a pan on cooking stove for 20 minutes”, we would not be able to distinguish the use of a frying pan and a cooking stove. Instead we should make 2 distinct steps: “put in frying pan” using a frying pan, and ”cook on stove

for 20 minutes". This works, albeit with some caveats which we will explain in detail in section 7.

An important thing to note is that ingredients are only specified in the steps where they are introduced. So if we use just one egg in a recipe, there is only one step that references that egg. So in the end we can tally the ingredients from all steps and get a shopping list.

An Ingredient can be substituted by another, for this purpose we added a relationship `CanBeSubstBy` between `CookingStep` and `IngredientCategory`. This is one of the reasons we disallow a step to have more than one ingredient.

An Ingredient has a unit field, this is the unit to be used with the `amount` referenced in a cooking step, despite what the description might employ. Handling unit conversions would therefore be left to the application's frontends. An ingredient object also includes information such as proteins, calories, and a price (per unit). An ingredient belongs to zero or more ingredient categories (such as grease, dairy, fermented). We make a clear distinction here, an ingredient can belong to several categories, but can only be substituted by ingredients from the category indicated by `CanBeSubstBy`. So in a recipe where you need a meltable cheese for example, the recipe might specify a specific meltable cheese from the categories cheese and dutch cheese. And then in the step it would indicate that the cheese may be replace by any other cheese from the meltable cheese category (but regular cheese).

In order to query for diet/allergies, one ingredient category can have zero or more restrictions. These restrictions are a generalisation of diets and allergies. They forbid one or more ingredient categories. For example, 'lactose intolerance' might forbid the use of any ingredients from the 'dairy' category.

3 Ontology

We chose to divide a recipe in a series of `CookingStep` because we'd like to have enough granularity to work with tools and substitute ingredients: one ingredient can be used multiple times in a recipe and might not serve the same role.

We chose to create a subclass of `CookingStep` called `CookingStepWithIngredient`. Only instances of the last type may add ingredients. We think there is a semantic difference between a set that introduces a new ingredient and other steps. The other steps could potentially be aggregated into one big step. The granularity of these steps is entirely up the the one entering the data. Another, more trivial, reason is because we specify the amount of an ingredient and its substitution categories on a step. And so because all steps that add an ingredient have to be of the `WithIngredient` type, we can easily check if the `amount` and `CanBeSubstBy` are present as well using SHACL, for example. It also makes it somewhat easier to quickly query all steps with ingredients for a recipe.

For the URIs of our ontology we chose `http://example.com/thelastrecipe/` as the base. To identify our resources we took their class name and appended their unique database identifier. So recipe with ID 1 can be referenced by

`http://example.com/thelastrecipe/recipe1`. We could have also used the name of the recipe, as in our ER diagram we indicated that the name would be unique. This would have been nice because it would have conveyed more semantic information to human users, but we have several reasons why we did not do this. The primary reason is that working with artificial identifiers is easier at the database level, and so it is easy to extend this to the ontology. In fact, at the moment we do not state that names are unique in our database schema. The secondary reason for using the artificial identifiers is consistency. While we could have used names for recipes, we could not have done so for cooking steps. Cooking steps are weak entities, even in our database they are identified by the recipe they belong to and their order. As such, a cooking step may be referred to as follows: `http://example.com/thelastrecipe/cookingstep2_8` (step 8 in recipe 2). It would have been a bit strange to use plain names for one class and identifiers for another. The last reason is just convenience. Imagine we used names for recipes for example and the name contained some arbitrary UTF-8 characters. These names would then have to be included in the URIs. We are not really aware of how Protégé could handle this, but the amount of times Protégé crashed on us does not inspire confidence. Because of this, we played it safe.

We used `xsd:string` instead of `xsd:token` because the latter is not supported by Ontop [1]. For the previous milestones we thought it was a better idea to use tokenized strings since it collapses white spaces and does not allow empty strings.

Similarly, we initially wanted to use `xsd:duration` to indicate how long a `CookingStep` would take. Conceptually it would have been the best choice but unfortunately neither Ontop nor R2RML seem to support the datatype. Thus we were forced to switch to `xsd:nonNegativeInteger`, indicating the number of minutes required.

4 Inferred rules

We came up with some inferred rules which provide data on recipes that can be inferred from the ingredients. Our first example is `isVegetarian`. This rule determines if a recipe is vegetarian by checking if it contains any ingredients that are forbidden for the 'vegetarian' restriction. This rule is but one example and can be implemented for every restriction in the ontology.

The `isVegetarian` (and similar diet-related properties) are an inferred properties that can be written as the following rule:

$$\forall recipe \forall s \forall i \forall c.$$
$$consistOf(recipe, s), hasIngredient(s, i), belongsTo(i, c), \neg forbids(c, \text{“}Vegetarian\text{”})$$
$$\rightarrow isVegetarian(recipe)$$

Finding an alternative ingredient consists of finding another ingredient that belongs to

[1] https://github.com/ontop/ontop/wiki/OntopDatatypes

one of the substitution categories.

$$\forall i' \forall cat. hasIngredient(step, i) \land belongsTo(i', cat) \land canBeSubstBy(step, cat) \rightarrow alternativeIngredient(s$$

We could infer optional ingredients in a receipt (e.g., grand-mother's secret ingredients) by introducing a special ingredient category such as "Nothing", any step that can be substituted by an ingredient of this category. No ingredient should belong to this category.

$$\forall i \forall s. hasIngredient(s, i), canBeSubstBy(s, Nothing) \rightarrow optionalIngredient(i)$$

The total price of a recipe can be inferred from the price per unit of each ingredient multiplied by the amount for each step. In a similar manner we can infer the total time a recipe takes to be prepared.

In the end we did not think these rules necessitated the use of a rule language. They can all be implemented using SPARQL, by just making use of path inferencing, and so we implemented them in SPARQL.

5 Mapping

For the mapping we used the Ontop Protégé plugin and the H2 database. We did not find any reason to diverge from the tools introduced in the course. We chose Ontop because our mappings are very straight-forward, it is a direct mapping. Ontop is also convenient (at least, in theory) because we can have all our data in the same place.

One thing worth mentioning might be that `CookingStepsWithIngredient` does not exist in our database, we map to a normal `CookingStep` table that might or might not reference an ingredient.

6 Demonstrator

For our demonstrator we implement some advanced queries. The scenario we envision is a website where users can search for recipes using various in-depth filters.

Unfortunately we encountered a problem while developing our demonstrator. We noticed too late that the Ontop SPARQL is seriously crippled. It does not support aggregate queries. Most of the advanced queries that are possible with our ontology require aggregates. For example to look up pricing, calorie or protein info or even to look up how long a recipe will take to prepare. Because of this we will provide a few simpler examples that will work on our endpoint, and a few more advanced ones that would work if only Ontop supported aggregates.

6.1 Basic example of SPARQL endpoint

This is an example of basic usage, where a user will retrieve information about a recipe.

```
PREFIX : <http://example.com/thelastrecipe/>
SELECT *
WHERE {
        ?id a :Recipe.
        ?id :name ?name.
        ?id :servings ?servings.
}
```

This is a slightly more advanced example that will get the amount of calories for each recipe.

```
PREFIX recipe: <http://example.com/thelastrecipe/>
SELECT ?name (SUM(?cals * ?amount) as ?calories)
WHERE {
  ?rec a recipe:Recipe.
  ?rec recipe:name ?name.
  ?rec recipe:ConsistsOf ?cookingstep.
  ?cookingstep recipe:HasIngredient ?ingredient.
  ?cookingstep recipe:amount ?amount.
  ?ingredient recipe:calories ?cals.
}
GROUP BY ?name
```

6.2 Filter examples

In this example the user filters for all recipes that do not have ingredients containing dairy.

```
PREFIX recipe: <http://example.com/thelastrecipe/>

SELECT (?r AS ?recipeName) WHERE {
    ?r a recipe:Recipe .
    ?r recipe:consistsOf ?step .
    ?step recipe:hasIngredient ?ingredient .
    ?ingredient recipe:belongsTo ?cat .
    ?cat recipe:name ?catName .
    FILTER(?catName != "Dairy")
}
```

6.3 Tool usage example

The user only wants recipes that require the use of an oven for less than 40 minutes.

```
PREFIX recipe: <http://example.com/thelastrecipe/>
SELECT ?name
WHERE {
    ?x a recipe:Recipe.
    ?x recipe:name ?name.
    {
        SELECT ?time
        WHERE {
            ?x recipe:consistOf ?s.
            ?s recipe:UsesTool ?t.
            ?t recipe:name "Oven".
            ?s recipe:timeRequired ?time.
        }
    }
    FILTER(?time < 40).
}
```

6.4 Advanced example

This is an advanced query that shows a user searching for a vegetarian recipe of the dinner category containing less than 500 calories that requires the oven for no more than 40 minutes and has a total price of less than 15 euros.

```
PREFIX recipe: <http://example.com/thelastrecipe/>
SELECT ?name
WHERE {
    ?x a recipe:Recipe .
    ?x recipe:name ?name .
    ?x recipe:isVegetarian true .
    ?x recipe:InCategory ?c.
    ?c recipe:name "Dinner".
    ?x recipe:price ?price .
    ?x recipe:calories ?calories  .
    ?x recipe:servings 1  .
    FILTER(?calories < 500) .
    {
        SELECT (SUM(?t) as ?ovenTijd) WHERE {
            ?x recipe:ConsistsOf ?step .
            ?step recipe:UsesTool recipe:Oven .
            ?step recipe:timeRequired ?t .
        }
    }
    FILTER(?ovenTijd <= 40 && ?price < 15.0).
}
```

7 Limitations

Our design has a couple of limitations, mostly related to the perhaps too simplistic design of the cooking step. A cooking step is only allowed to have 1 ingredient and 1 tool. Recipes you can find online often include 3 or 4 ingredients in one step. It could be something like "Add ingredient A, B, C into the pot and blend". In our design this is not possible, and the step has to be split up into a step for every ingredient, adding them to the pot one step at a time.

Added to this is the restriction of 1 tool per step. Because we do not allow for concurrent steps, we cannot accurately model more than 1 tool being in use at the same time. A common example is "Heat a skillet for 5 minutes". We actually use 2 tools in this step: a furnace, and a skillet. Because we cannot represent this in our model, we are forced to have 1 step to "take out a skillet" that takes no time and uses the skillet, and then a step "heat the skillet on the furnace" that takes time and uses the furnace. A consequence of this is that we cannot really query for recipes that use a tool for some amount of time. We have tried to remedy this somewhat by adopting the convention that appliances take priority. This means that we will always allocate the time to the furnace for example, instead of the skillet. This makes the most sense as most people probably do not care how long they have to use a skillet, it is the furnace that matters.

These restrictions are present in our database design, and so it is not really possible to retrofit a solution in the ontology. The result of these restrictions is that we have somewhat reduced expressivity in our model. Adapting recipes to our model is a very time-consuming task and so we are not able to sample a representative amount of recipes to check for the precise impact of our restrictions. However, from our experience adapting a few recipes, we find that we need about twice as many steps as a regular written-down recipe.

8 Teamwork

Most of the work for this project was done as a group. For every deadline except the last one we had a meeting at the VUB or through voice chat. Table 1 shows the distribution of the tasks we did individually.

9 Conclusion

As mentioned before, we had to restrict the use of only one tool in each cooking step. We could improve this by replacing

Our system does not handle concurrent steps very well: we do not have a way to indicate that different cooking steps happen at the same time (e.g., preparing vegatables while the meat is baking). For everything to function correctly, the client must set the `timeRequired` to zero on these concurrent steps otherwise the total time required of the recipe would be overestimated.

Assigned to	task
Aaron	Digital conceptual schema
Khaïm	Digital inferring rules
Kelvin	Database schema
Kelvin	Creating Ontology in Protégé
Kelvin	Creating mappings in Protégé
Aaron	Data conversion and entry
Stijn	Schema fixing and data cleanup
Stijn	SHACL constraints
Khaïm & Aaron	Report

Table 1: Specific task distribution

One easy improvement to our system would be to simply add more information, such as other attributes to Ingredients comparable to proteins. Similarly ingredient categories could also benefit from more information which would allow encoding properties for entire groups of ingredient.

A SHACL

We implemented SHACL constraints for our ontology. This is especially useful for our ontology because otherwise certain situations would be possible that make no sense. For example you could have a cooking step with 2 `amounts`. Or a cooking step with `amount` but no actual ingredients attached. Because of this we check for cardinality on all properties. We also have some additional constraints like for example that the name of a category cannot be an empty string. Or that an amount cannot be negative. This last one especially we cannot really check with just OWL. We have to use the datatype `xsd:double` for it because you could have something like "half a cup of milk" in a recipe, and there is no `nonNegativeDouble`.

Apart from this we also use SHACL to check if all our properties are labeled and documented. We realize this is overkill for an ontology of this size, but we were using SHACL for cardinality checking anyway. The SHACL file is quite large so it is not included in this document. It can be found in our repository.

B Visualizations

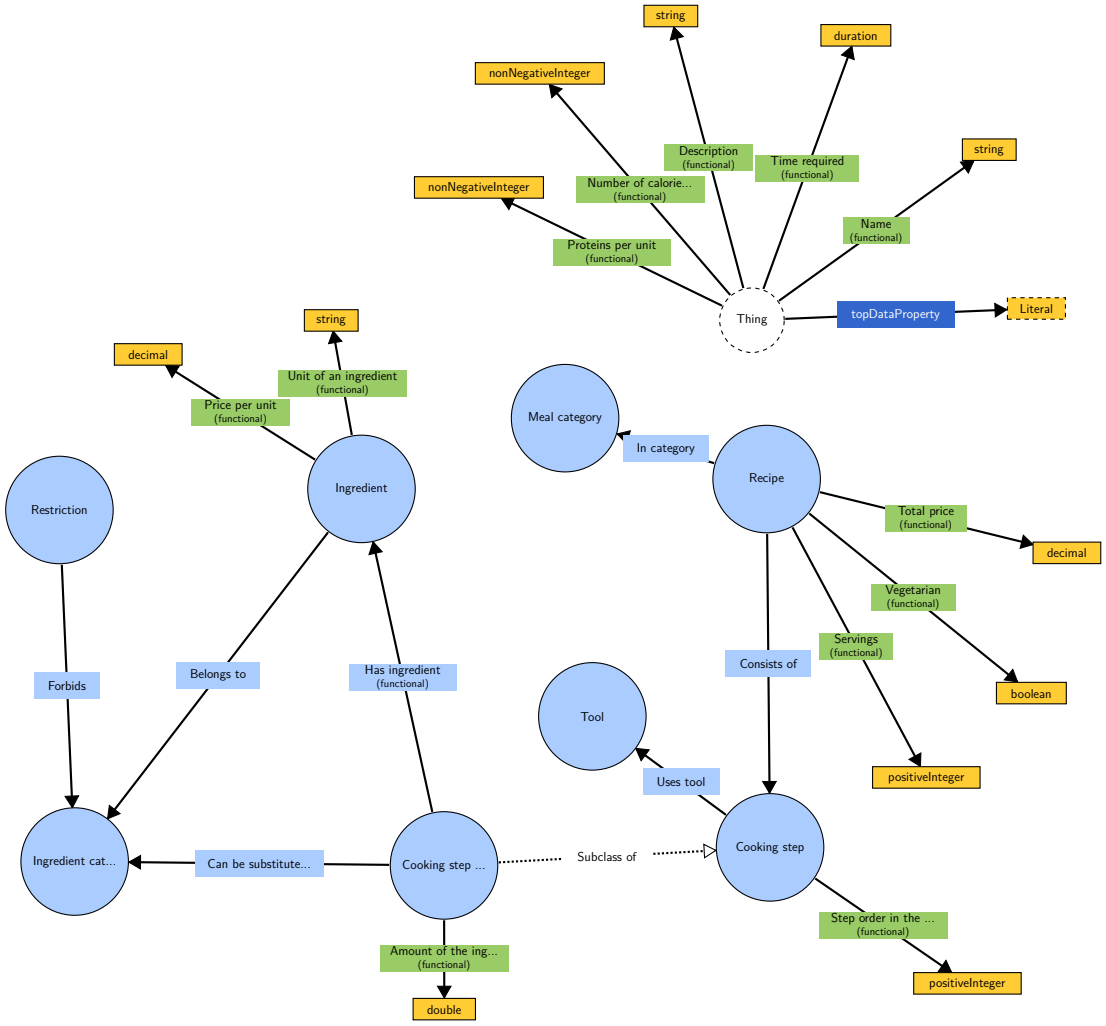

Figure 1: WebVOWL Visualization of the ontology

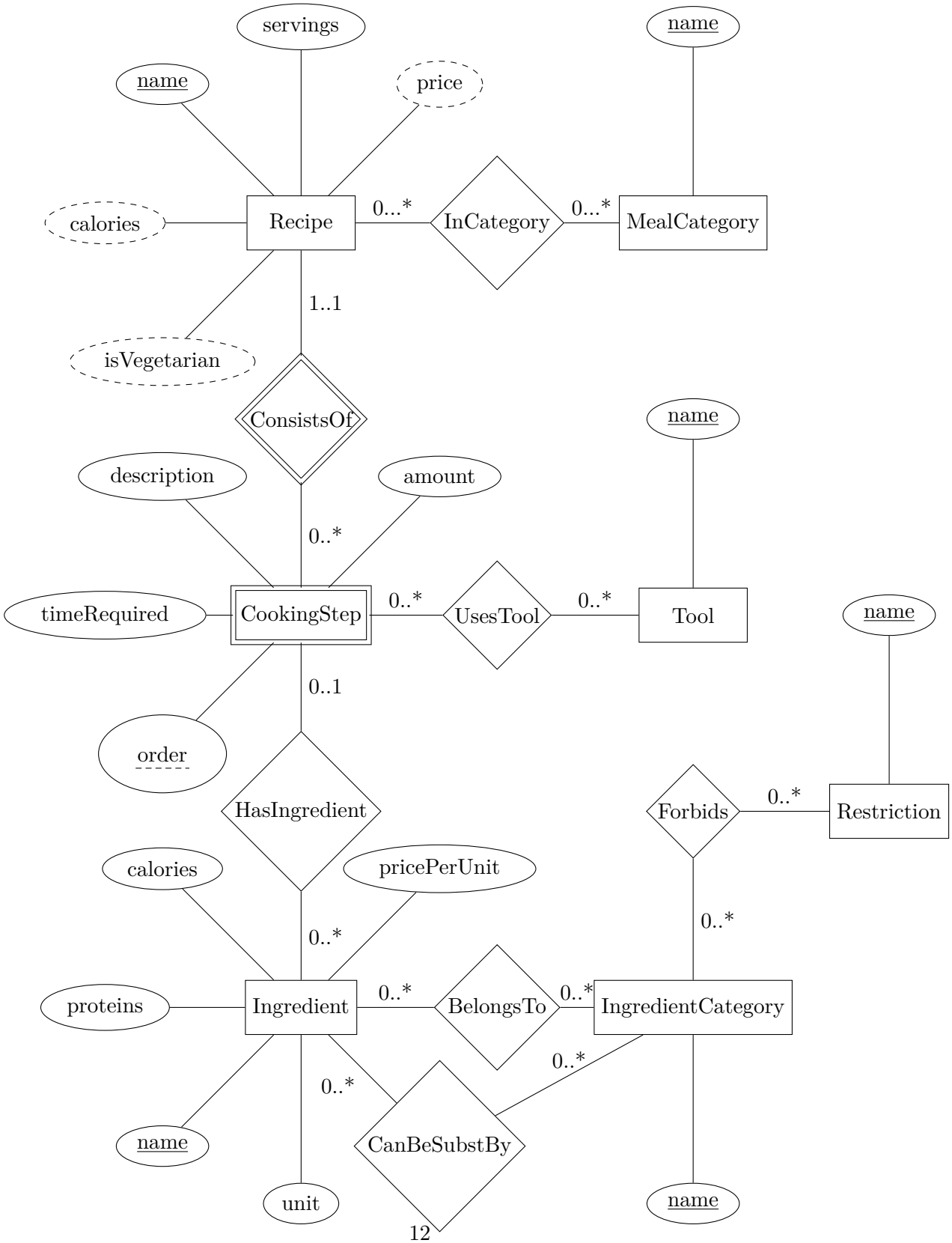

Figure 2: ER Diagram

References

[1] Khaïm Berkane, Aaron Lippeveldts, Kelvin Schoofs, and Stijn Vissers. *Github Repository containing all artifacts.* `https://github.com/svissers/Open-Information-Systems-Project`, 2019.

[2] Fajar Ekaputra and André Wolski. *SHACL4P Plugin - SHACL Constraint Validation plugin for Protégé Desktop* . `https://github.com/fekaputra/shacl-plugin`, 2016. [Online; accessed 18-December-2019].

[3] Rafael Gonçalves, Josef Hardi, Matthew Horridge, Samson tu, and Mark Musen. *Protégé Desktop v5.5.0.* `https://protege.stanford.edu/about.php`, 2016. [Online; accessed 22-November-2019].

[4] Vincent Link, Steffen Lohmann, Eduard Marbach, Stefan Negru, and Vitalis Wiens. *WebVOWL: Web-based Visualization of Ontologies.* `http://www.visualdataweb.de/webvowl/`, 2014. [Online; accessed 22-November-2019].